BEHAVIOR-BASED ATTACK GENERATION FOR DETECTING WEB APPLICATION VULNERABILITIES

Thesis

Submitted in partial fulfilment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

by

G. DEEPA

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

NATIONAL INSTITUTE OF TECHNOLOGY KARNATAKA

SURATHKAL, MANGALORE - 575 025

June, 2018

DECLARATION

by the Ph.D. Research Scholar

I hereby declare that the Research Thesis entitled **Behavior-based Attack Generation for Detecting Web Application Vulnerabilities** which is being submitted to the **National Institute of Technology Karnataka, Surathkal** in partial fulfilment of the requirements for the award of the Degree of **Doctor of Philosophy** in Department of Computer Science and Engineering is a bonafide report of the research work carried out by me. The material contained in this Research Thesis has not been submitted to any University or Institution for the award of any degree.

G. Deepa, CS13F06
Department of Computer Science and Engineering

Place: NITK, Surathkal.
Date: June 07, 2018

CERTIFICATE

This is to certify that the Research Thesis entitled **Behavior-based Attack Generation for Detecting Web Application Vulnerabilities** submitted by **G. Deepa** (Register Number: 138032 CS13F06) as the record of the research work carried out by her, is accepted as the Research Thesis submission in partial fulfilment of the requirements for the award of degree of **Doctor of Philosophy**.

Dr. P. Santhi Thilagam
Research Guide

(Signature with Date and Seal)

Chairman - DRPC
(Signature with Date and Seal)

To my parents Mr. Ganesan (late) & Mrs. Saratha, and my sister Sarala.

ACKNOWLEDGEMENTS

I take this opportunity to express my sincere gratitude to all the people who encouraged me to finish this dissertation.

First and foremost, I would like to thank my supervisor and Head of the Department, Dr. P. Santhi Thilagam, for providing me an opportunity to pursue PhD under her guidance. I appreciate her continuous support, encouragement, and recommendations in both professional and personal aspects of my life.

I express my sincere thanks to the members of my research progress assessment committee, Dr. A. Senthil Thilak and Dr. Basavaraj Talawar, for their positive criticism and insightful comments on my research work. I would also like to thank Dr. Alwyn R. Pais for his suggestions and support to improve the quality of the work. I also thank the faculty and staff members of the department for their support.

This work is a part of the Research & Development project supported by the Ministry of Electronics and Information Technology (MeiTY), Government of India. The project provided me the opportunity to pursue my research, and I would like to thank the team members of the project, Amit Praseed, Nushafreen Palsetia, Furqan Ahmed Khan, Prabhjeet Kaur and Ramesh Babu, for providing me necessary technical assistance.

Special thanks are due to Bindu, who constantly supported me during the tenure of my PhD. I would also like to thank Dr. Ganesh K Reddy, Dr. Raghavendra Achari, Fathima, Shiva Kumar, and Ambikesh for their help extended during the course of present study.

I am deeply indebted to my parents, sister and brother-in-law for being with me all through, and encouraging me for the successful completion of the research work. Thanks to my husband, friends and family members for their support.

G. Deepa

ABSTRACT

Web applications provide a convenient platform to support a wide range of day-to-day activities such as bill payments, online shopping, banking, and social networking. However, the accessibility, omnipresence, demand, and ever-growing user-base have made web applications an attractive target for attackers. The attacks on web applications occur due to the existence of weaknesses in the applications, which allow the attackers to exploit and perform adverse actions. These weaknesses are known as vulnerabilities, and are broadly categorized as *Injection vulnerabilities* and *Logic vulnerabilities*, which are rated as the most potent vulnerabilities by different security consortiums. Hence, in order to secure web applications from the attacks, it is indispensable to detect these vulnerabilities. The vulnerabilities in web applications are detected using either white-box or black-box analysis. While the former analyzes the source code of the application, the latter penetrates the application with malicious inputs/requests, and observes the output for exposing the vulnerabilities. The primary challenge during the penetration lies in producing malicious requests automatically based on the intended behavior of the application. Therefore, this work aims at developing a behavior-based approach to generate attack requests through black-box analysis for the detection of logic and injection vulnerabilities in web applications.

Logic vulnerabilities in web applications allow the malicious users to compromise the application-specific functionality against the expectations of the stakeholders. These vulnerabilities are introduced due to missing/incorrect server-side validation, access checks, and sequence checks, and are known as parameter manipulation, access-control, and workflow vulnerabilities respectively. Logic vulnerabilities are application-specific, and hence detection of these vulnerabilities through black-box analysis is extremely challenging as it requires a clear understanding of the intended behavior of the application for generation of attack requests. The intended behavior can be inferred by examining both the data flow and control flow information of the application. The existing approaches utilize either the data flow or control flow to infer the intended behavior, and are capable of detecting only a specific type of logic vulnerability. Hence, there is a demand for a system that is capable of inferring the intended behavior of the application in order to generate attack requests for detecting all types of logic vulnerabilities. The proposed work aims at modeling the intended behavior of the application in the form of an annotated Finite State Machine (FSM) using both the data flow and control flow information obtained from web application traces. The constructed model is utilized to generate attack requests for identifying all types of business logic vulnerabilities. The constructed model is evaluated on vulnerable benchmark applications, and the experimental results substantiate the effectiveness of the proposed model in comparison with the recent approaches. In addition, the model helps in detecting logic vulnerabilities leading to session puzzling attacks, which is not addressed in the existing approaches.

Web applications store data in relational databases traditionally. However, a lot of web applications in use today are XML-based as they involve exchange of information through XML documents, and store these documents in Native XML Databases (NXDs). NXDs are generally preferred for applications that hold highly diverse information, involve integration of information from different set of applications, handle rapidly evolving schemas, and work with a huge set of documents or large-sized documents (e.g., books, web pages). The existing literature assures the growing demand towards usage of NXDs. The database of the applications is targeted by attackers to inject code fragments into user-input which attempt to modify the query submitted to the database resulting in SQL/XML injections. While SQL injection targets relational databases, XML injection targets NXDs that utilize XQuery/XPath as the query language. The literature available to address XML/XPath/XQuery injections is relatively less compared to SQL injection, and the existing approaches for vulnerability detection focus on manual construction of individual attack requests based on known types of attacks. Therefore, there is a demand for an approach that enables the generation of attack requests for detection of all types of XQuery injection vulnerabilities specified by the security consortium-Open Web Application Security Project (OWASP), and that is extendable enough to generate attack requests leading to unknown types of attacks. Hence, this work formulates an attack grammar for generation of attack requests to identify XQuery injection vulnerabilities in web applications driven by NXDs. The strings generated by the attack grammar are injected into the web application for detecting the vulnerabilities. In addition to the different types of attacks listed by OWASP, three new categories of XQuery injection attacks namely *alternate encoding, injection through evaluation function*, and *XQuery comment injection* attacks are discovered. These attacks demonstrate the extendability of the proposed attack grammar. The proposed grammar is evaluated on vulnerable benchmark applications from the test suite of AMNESIA. The experimental results substantiate the effectiveness of the proposed grammar in detecting all types of XQuery injection vulnerabilities.

Keywords: Web security, Database security, Security testing, Vulnerability discovery, Penetration testing, Application logic vulnerabilities, Logic attacks, Parameter tampering, Application flow bypass.

CONTENTS

List of Figures **vii**

List of Tables **ix**

List of Algorithms **xi**

List of Abbreviations **xiii**

1 Introduction **1**

1.1 Web Application Architecture 1

1.2 Web Application Threats 3

1.3 Web Application Security 7

1.4 Motivation 9

1.5 Thesis Contributions 11

1.6 Thesis Organization 12

2 Literature Review **13**

2.1 Web Application Vulnerabilities 13

2.1.1 Injection Vulnerabilities 15

2.1.2 Business Logic Vulnerabilities 17

2.1.3 Session Management Vulnerabilities 19

2.2 Defensive Mechanisms for Securing Web Applications 19

2.2.1 Injection Defenses 19

2.2.1.1 SQL Injection 19

2.2.1.2 XML Injection 24

2.2.1.3 Cross-Site Scripting 29

2.2.1.4 Research Challenges and Directions 36

2.2.2 Session Management Defenses 41

2.2.3 Business Logic Defenses 43

2.2.3.1 Parameter Tampering 43
2.2.3.2 Access-Control Violation 46
2.2.3.3 Workflow Violation 51
2.2.3.4 Business Logic Preservation in eCommerce Applications 53
2.2.3.5 Research Challenges and Directions 57
2.2.4 Existing Vulnerability Scanners 58
2.3 Research Gaps 61
2.4 Summary 62
3 Problem Description 63
4 Attack Generation for Detecting Logic Vulnerabilities 67
4.1 Logic Attacks in Web Applications 68
4.2 Problem Description 71
4.3 Proposed Approach 78
4.3.1 Trace Collection 79
4.3.2 Model Construction 80
4.3.3 Intended Behavior Extraction 85
4.3.4 Attack Generation 93
4.4 Experimental Study and Analysis 101
4.4.1 Experimental Setup 101
4.4.2 Test Applications 104
4.4.3 Experimental Results and Discussions 105
4.4.4 Advantages and Limitations 109
4.5 Summary 111
5 Attack Generation for Detecting XQuery Injection Vulnerabilities 113
5.1 XQuery Injection in Native XML Database 114
5.2 Problem Description 118
5.3 Proposed Approach 118
5.3.1 Identification of Injection Points 119
5.3.2 Attack Generation Based on OWASP Guidelines 119

5.3.3 Identification of XQuery Injection Attacks not Listed in OWASP 123

5.3.3.1 Alternate Encoding Attack 123

5.3.3.2 Injection Through Evaluation Function 125

5.3.3.3 XQuery Comment Injection Attack 127

5.4 Experimental Study and Analysis 128

5.4.1 Experimental Setup 128

5.4.2 Test Applications 130

5.4.3 Experimental Results and Discussions 130

5.4.4 Advantages and Limitations 133

5.5 Summary 134

6 Conclusions and Future Scope 135

6.1 Summary of the Contributions 135

6.2 Future Scope 137

Bibliography 139

Publications 165

LIST OF FIGURES

1.1 Web application architecture 2
1.2 Security mechanisms for protecting web applications 7

2.1 Types of vulnerabilities and attacks 14
2.2 Research on detection and prevention of XSS vulnerabilities 34
2.3 Research on detection and prevention of XSS attacks 37
2.4 Research on parameter manipulation vulnerabilities 46
2.5 Research on access-control vulnerabilities 52
2.6 Research on workflow vulnerabilities 54

4.1 Web application model 82
4.2 Annotated finite state machine for application Scarf 86
4.3 Annotated finite state machine for application Wackopicko 87
4.4 Workflow graph 91
4.5 DetLogic architecture 102

5.1 XQueryFuzzer architecture 129

LIST OF TABLES

1.1 Data breaches in the recent years 4

2.1 Summary of SQL injection prevention/detection approaches 25
2.2 Summary of XML injection prevention/detection approaches 30
2.3 Summary of articles on XSS 38
2.4 Summary of the literature on logic vulnerabilities 55
2.5 List of commercial and open-source scanners and their capabilities . . . 59

4.1 Symbols and definitions 71
4.2 Bugs and corrections 92
4.3 Applications for evaluation 104
4.4 Parameter manipulation vulnerabilities detected 106
4.5 Number of constraints extracted 106
4.6 Access-control vulnerabilities detected 106
4.7 Types of access-control vulnerabilities detected 107
4.8 Comparison of the proposed approach with LogicScope 108
4.9 Workflow vulnerabilities detected 109

5.1 Test application details 130
5.2 XQuery injection vulnerabilties detected 131
5.3 Number of vulnerabilities existing in test applications and detected by the proposed approach 132
5.4 Effectiveness of the proposed approach 132
5.5 Comparison of the proposed approach with ZAP 133

LIST OF ALGORITHMS

4.1 Model construction 84

4.2 Parameter manipulation attack generation 94

4.3 Authentication bypass attack generation 95

4.4 Vertical privilege escalation attack generation 96

4.5 Horizontal privilege escalation attack generation 98

4.6 Session puzzling attack generation 99

4.7 Workflow bypass attack generation 100

LIST OF ABBREVIATIONS

Abbreviations	Expansion
ACM	Access-Control Mechanism
ACP	Access-Control Policy
ACT	Access-Control Template
ACV	Access-Control Violation
AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
ASP	Active Server Pages
BP	Base Pointer
CSRF	Cross-Site Request Forgery
CSS	Cascading Style Sheet
CVSS	Common Vulnerability Scoring System
CWE	Common Weakness Enumeration
DFA	Deterministic Finite Automata
DOM	Document Object Model
EAR	Execution After Redirection
EFSM	Extended Finite State Machine
EHR	Electronic Health Record
EJB	Enterprise Java Beans
ELET	Embedded Language Encapsulation Type
FARM	Fuzzy Associative Rule Model
FNR	False Negative Rate
FPR	False Positive Rate
FSM	Finite State Machine
GUI	Graphical User Interface
HPP	HTTP Parameter Pollution
IDE	Integrated Development Environment
IDS	Intrusion Detection System
IP	Instruction Pointer
JSP	Java Server Pages
LDAP	Lightweight Directory Access Protocol
LOC	Lines Of Code
NDFA	Non-Deterministic Finite Automata
NXD	Native XML Database

Abbreviations	Expansion
OWASP	Open Web Application Security Project
PCI DSS	Payment Card Industry Data Security Standard
PHP	Hypertext Preprocessor
PM	Parameter Manipulation
QR	Query Restrictor
RIA	Rich Internet Application
SDLC	Software Development Life Cycle
SEPTIC	SElf-Protecting daTabases preventIng attaCks
SMV	Session Management Vulnerability
SOFIA	Security Oracle For Injection Attacks
SQL	Structured Query Language
SQLIA	SQL Injection Attack
SQLIV	SQL Injection Vulnerability
SWAP	Secure Web Application Proxy
TBRC	Tibetan Buddhist Resource Center
TNR	True Negative Rate
TPR	True Positive Rate
UA	User Authenticator
WASC	Web Application Security Consortium
WFB	Workflow Bypass
WVS	Web Vulnerability Scanner
XML	Extensible Markup Language
XSS	Cross-Site Scripting
ZAP	Zed Attack Proxy

CHAPTER 1

INTRODUCTION

Over the years, web applications have 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 application is an application program that resides on a server and makes use of a browser on the client machine to provide service to the end-user. It allows the website visitors to interact with the application server through their user interface, and to submit requests for retrieving data to/from the database over the Internet using their preferred browser. Precisely, the web applications query the database server to generate web pages dynamically for presenting to the client. With the advancements in the Internet, network and technology, many organizations are moving their services from offline to online, which made web applications so popular and an integral part of the daily life. Hence, web applications play a significant function in mission-critical areas to perform financial transactions, social communications, online shopping, and so on, and often handle sensitive user data. The complexity of the web applications increases with number of features offered owing to emerging technologies.

1.1 WEB APPLICATION ARCHITECTURE

With more than three billion users using the Internet for sensitive operations,[1] the technologies involved in the evolution of web applications have also seen a tremendous growth.[2] The three-tier architecture of the web application, the softwares used at each-

[1]http://www.internetworldstats.com/stats.htm
[2]http://www.evolutionoftheweb.com/#/evolution/day

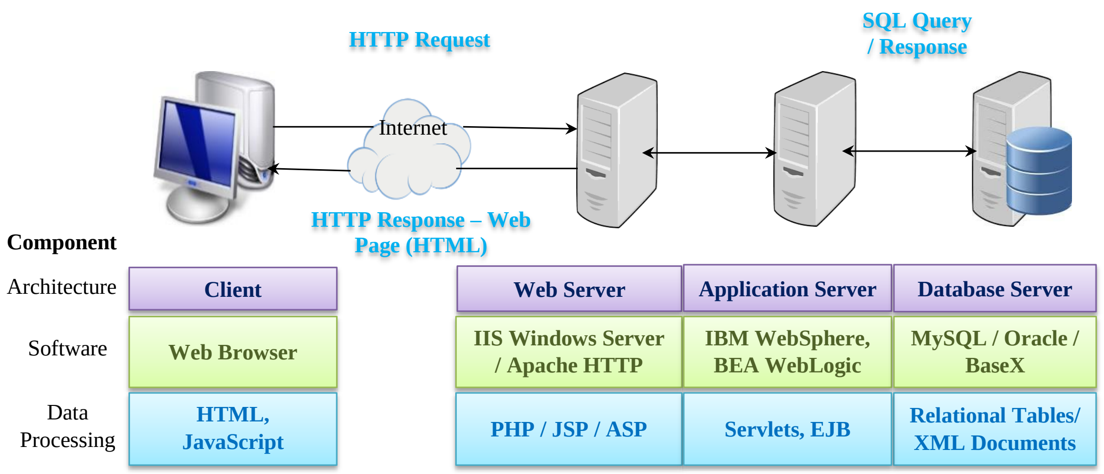

Figure 1.1: Web application architecture

tier and the technology involved in processing the data being transferred is illustrated in Figure 1.1 and is explained as follows:

Client-side: The client/user triggers an HTTP request to the web application using a web browser. The web browser loads the HTML web pages formatted using stylesheets (e.g., CSS) along with the client-side scripts (e.g., JavaScript, VBScript) received from the web server. The web pages constitute the Graphical User Interface (GUI) of the application through which the end-user interacts with the application. The client-side scripts constitute the presentation logic (e.g., input validation), and increase the responsiveness of the application.

Server-side: The server-side comprises of the web server and the application server. The HTTP request from the browser is forwarded to the web server which processes the request using server-side scripts written in any one of the following: JSP, ASP, PHP, etc. Apache HTTP Server, IIS Windows Server are few examples of web servers. The web server serves the content using HTTP/HTTPS protocol. The application server (IBM WebSphere, etc.) includes the processing logic of the application implemented in Enterprise Java Beans (EJB), serves content using HTTP/HTTPS as well as other protocols, and provides support to web services. Web services is a component of the web application used for exchanging data between different web applications running in different platforms.

The processed request is forwarded to the database server for retrieving/inserting data from/into the database, where data is stored in the form of relational tables/XML (Extensible Markup Language) documents. Relational databases such as MySQL/Oracle are used when the data associated with the application follows the relational model, and native XML databases like BaseX are used when the application deals with rapidly evolving schema (Bourret 2009). The data retrieved from the database is then subjected to further processing, and is rendered back to the client, which can be viewed in the browser as an HTML page formatted using CSS and JavaScript for enhancing the behavior.

1.2 WEB APPLICATION THREATS

Web applications have become a prime target for attackers due to their ease of use, omnipresence, demand and evergrowing user-base. The security breach reports from various organizations signify the importance towards securing web applications. According to Symantec Internet Security Threat report (Symantec 2016), one million attacks were reported on web applications in 2015, and 75% of legitimate applications had unpatched vulnerabilities. A report from Trustwave (2016) states that 97% of the applications tested by Trustwave had security vulnerabilities, and a median of 14 number of vulnerabilities is discovered per application. A report by the Identity Theft Resource Center (ITRC 2015) 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. According to Verizon's Data Breach Investigation Report (Verizon 2014), 35% of the security incidents in 2013 were due to attacks on web applications. Table 1.1 lists the details of the data breaches and types of web application attacks reported in the recent years.

Web applications are more prone to various attacks due to the advancements in architecture and technologies that provide sophisticated functionalities and therefore increase the complexity of the web application. The evolving technologies fail to consider security of the application due to the following factors:

Table 1.1: Data breaches in the recent years

Year	Company	Data Breach	Attack
2017	Gmail (Daitch 2017)	Unauthorized access to accounts of 1 million Gmail users	Phishing
2016	UN Tourism website (Murdock 2016)	Usernames, email addresses and hashed passwords of 1,300 users	SQL injection
2016	Illinois State Board of Elections voter database (Sweet 2017)	Personal information of 80,000 users	SQL injection
2015	World Trade Organization (Paganini 2015)	Personal information of 53,000 users	SQL injection
2014	eBay (Muncaster 2014)	Login information of active users	Cross-site scripting
2012	Domino's India (Dominos 2012)	Personal information of 37,000 users	SQL injection and Remote file inclusion
2012	Yahoo (Keizer 2012)	Login information of 453,000 users	SQL injection
2011	Citi Credit Card (Zetter 2011)	Financial data exposure of 360,000 customers	Parameter Tampering
2010	AT&T website (Acunetix 2010)	Email addresses of 100,000 Apple customers	Parameter Tampering

i) 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) 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 implementation.

iii) The interoperability and openness of XML used for providing interaction between heterogeneous web applications make them an easy target for attackers.

iv) Construction of web applications by developers focusing on implementing the features and functionality of the application rather than the security aspects.

As a result, the existing web applications are more vulnerable to attacks that compromise the confidentiality, integrity, and availability of data.

As stated already, the data breaches occur due to the propagation of attacks through the weakness in the application itself rather than the weakness in the network. These weaknesses are referred to as software security vulnerabilities. A vulnerability is a weakness in the application arising due to an implementation bug or a design flaw in the programming language, application development framework, architecture and code library (i.e., APIs), etc., and allows the attackers to cause damage to the application (Liu and Cheng 2009). These vulnerabilities could be exploited by injecting malicious code into input supplied by a user for interacting with the application. The malicious code in the input propagates through the application due to the existence of the following coding flaws such as improper input validation, improper authentication and authorization mechanisms, improper management of session information, and other implementation bugs compromising the intended functionality of the application (Howard et al. 2009; Igure and Williams 2008; Meunier 2008; Tsipenyuk et al. 2005).

Precisely, a security flaw/weakness can be defined as a defect in a software application or component that can lead to a software vulnerability when combined with the necessary conditions. A vulnerability can be stated as a set of conditions that allows violation of an explicit or implicit security policy. The different types of security flaws and the vulnerabilities arising due to these flaws are explained below.

Improper input validation refers to absence of validation or erroneous validation of input supplied by a user through user 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 lead to *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 sequence of operations* 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 *improper input validation* at server-side allows injection of input that violates the semantic restrictions on user-input, and hence allows violation of the business specifications. The above three implementation flaws lead to *Business Logic Vulnerabilities*. A report by Trustwave (2011) describes logic flaws as the second topmost threat, and these flaws are gaining attention of the researchers in the recent past.

Improper session management pertains to weakness in generation and handling of session tokens, which are necessary for maintaining the identity of end-user of the application and mapping relationship between consecutive requests of the application. These flaws allow the attacker to compromise the session of a valid user and perform adversarial actions, and lead to *Session Management Vulnerabilities*.

Each security flaw or weakness in web applications is assigned a unique number called Common Weakness Enumeration (CWE)[3] for easier tracking and understanding. CWE is a community-developed dictionary of software weakness types, which provides a unified and measurable set of software weaknesses. The CWE identifiers associated with the aforementioned implementation flaws are listed as follows. CWE-20 is related to improper input validation, and CWE-596 is related to improper semantic validation of input. CWE-840 is related to business logic errors; CWE-306, CWE-862, CWE-250, CWE-863, and CWE-732 are related to incorrect implementation of authentication and authorization mechanisms; and CWE-841 is related to improper enforcement of intended workflow of the application. CWE-613 and CWE-384 are weaknesses related to improper session management.

The conventional security measures such as Secure Socket Layer and cryptographic techniques 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 applications (Sima 2003). Hence, there is a need to protect the applications from attacks due to exploitation of vulnerabilities at the application layer, and organizations such as Open Web Application Security Project (OWASP) and Web Application Security Consortium (WASC) emphasize on protecting web applications from attacks due to exploitation of vulnerabilities.

[3]`https://cwe.mitre.org/`

Considering the impact, severity, frequency of attacks, and the focus of existing research works (Garcia-Alfaro and Navarro-Arribas 2008; Halfond et al. 2006; Hydara et al. 2015; ITRC 2015; Li and Xue 2014; OWASP 2016; SANS 2011; Scholte et al. 2012a; Shahriar and Zulkernine 2012b; Symantec 2014; Top10 2013; Trustwave 2011, 2014; Verizon 2014; Whitehat 2014), the scope of this work is limited to Injection and Business Logic Vulnerabilities.

1.3 WEB APPLICATION SECURITY

Web application security is a concept developed for securing web applications from malicious users by preventing attacks. In order to develop proactive web applications that safeguard from attacks, it is essential for the developers to incorporate several precautionary measures during the construction, testing, and post-deployment (i.e., runtime protection) of the application as stated in the literature (Antunes and Vieira 2012; Hein and Saiedian 2009; Microsoft 2015; Win 2014). Figure 1.2 shows a clear picture of the processes carried out for protecting web applications from attacks, and based on these processes the state-of-the-art approaches are classified as follows.

(a) Secure Construction/Programming: This technique minimizes the probability of attacks on web applications by allowing the programmers to follow *defensive coding* practices and guidelines during development of the application (OWASP 2017a,b). The secure coding practices help in preventing critical vulnerabilities in the application.

(b) Vulnerability Detection & Prevention: Even though various defense mechanisms have been proposed to prevent application attacks, the developers fail to enforce the

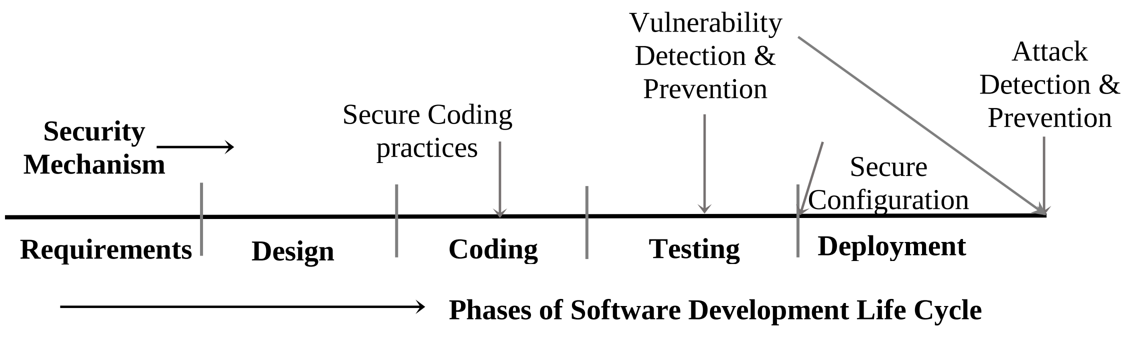
Vulnerability Detection & Prevention
Attack Detection & Prevention
Security Mechanism
Secure Coding practices
Secure Configuration
Requirements
Design
Coding
Testing
Deployment
Phases of Software Development Life Cycle

Figure 1.2: Security mechanisms for protecting web applications

security mechanisms during construction due to the lack of knowledge about the security (Scholte et al. 2012a). Therefore, it becomes necessary to provide a next layer of defense to the application security before deployment. Vulnerability detection is one such mechanism that analyzes the application for uncovering security vulnerabilities during early stages of deployment (Cova et al. 2007b). The two detection techniques available for identifying vulnerabilities are *static analysis* and *dynamic analysis*, and are explained below.

Static Analysis/White-Box Analysis: It examines the source code of the web application and explores all possible program paths for finding flaws. 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 Analysis: It penetrates the web application with malicious inputs with an intention of breaking the application, and identifies vulnerabilities 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, it cannot guarantee the precision and completeness as it does not explore all possible program paths of the application.

The advantage of vulnerability detection mechanism is the feasibility in employing the same for analyzing and testing applications which are in the process of development as well as legacy applications. The detected vulnerabilities should be eliminated before deployment to protect the application. Recent research works are inclined towards repairing the source code of the application in an automated fashion to eliminate vulnerabilities, and this mechanism is referred to as Vulnerability prevention.

(c) Attack Detection & Prevention: Attack detection and prevention systems are the final step of defenses that can be applied for web applications under operation. These systems are either placed as proxies intercepting the requests between the client and server or involved in instrumentation of the application under test for preventing the attacks. Therefore, these systems degrade the performance of the application.

The aforementioned classification of the existing approaches shows that the security aspects of an application should be considered during the entire period of web application development to defend the application against attackers.

1.4 MOTIVATION

The defensive mechanisms suggested for securing web applications have their own strengths and weaknesses. Secure construction of applications is intended for application in the process of development, and hence cannot protect legacy applications from attacks. The vulnerability detection techniques employing white-box approach demand the source-code of the application and are technology dependent. These techniques do not require execution of the application, and hence cannot discover flaws that can be identified by observing the runtime behavior of the application. Black-box detection of vulnerabilities requires testing the application with malicious inputs, which is a crucial phase during identification. The problems with attack prevention techniques are generation of false positives that block legitimate transactions from being executed, and instrumentation of the source code for analyzing the HTTP packets. While secure construction and attack prevention techniques are well-suited for applications under development and operation respectively, vulnerability detection techniques are suited for both applications under construction and legacy applications. Therefore, vulnerability detection mechanism plays an inevitable role in securing web applications.

Vulnerability detection in web applications has been well-researched in the recent decade and a multitude of approaches have been proposed. White-box detection of vulnerabilities analyzes the source code of the application to discover coding flaws, whereas black-box detection of vulnerabilities requires execution of the application under test. Black-box detection of vulnerabilities generally operates in three phases: (i) it learns the behavior of the application under normal circumstances, (ii) it penetrates the application with malicious inputs, and (iii) finally it compares the obtained response with normal response to discover the flaws. Among the three phases, penetration testing the application with malicious inputs is a crucial phase for effective identification of vulnerabilities. Penetration testing is devoted to stressing the application with respect to the security features for detecting vulnerabilities. Hence, this work is inclined towards appropriate generation of malicious inputs and attack requests/vectors for discovering vulnerabilities in the application under test.

As web applications get increasingly complex to support sophisticated business functionalities, logic and injection vulnerabilities have attracted increasing attention in recent years as stated in Section 1.2. Therefore, this work concentrates on appropriate generation of attack vectors for detecting logic and injection vulnerabilities. The first part of this thesis focuses on logic vulnerabilities as logic flaws are most often driven by financial motives. A report by Trustwave (2016) states that 64% of the tested applications had session management vulnerabilities, 41% had server-side vulnerabilities, and 39% of them had authentication and authorization vulnerabilities. This signifies the importance towards detecting logic vulnerabilities. Unlike injection vulnerabilities, logic vulnerabilities are often difficult to find and anyone without an understanding of how the application is supposed to function would be oblivious to their existence and potential impact. These vulnerabilities cannot be detected through scanning or by using any type of Intrusion Detection System (IDS) or Firewalls. Even though large body of work exists for addressing injection vulnerabilities, there exists only limited number of works addressing logic vulnerabilities as the detection requires a clear picture regarding the behavior of the application. This motivated us to develop an approach for generation of attack vectors based on the behavior of the web application for identifying different types of logic vulnerabilities in the application.

The second part of the thesis focuses on generation of attack vectors for detecting XQuery injection vulnerabilities in applications driven by native XML databases. According to the Payment Card Industry Data Security Standard (PCI DSS) and Common Vulnerability Scoring System (CVSS), XQuery and XPath injections are high risk threats in addition to SQL injection, and hence detection of the vulnerabilities that could lead to these injection attacks is critically important (Gordeychik 2008). Traditional web applications generally use relational databases such as SQL, while Extensible Markup Language (XML) came into focus after the rise and growth of the Internet. XML is a data representation that favors integration and interoperability between heterogeneous web applications. The information exchanged between the applications in the form of XML documents can be processed efficiently when they are stored appropriately. These documents are stored in either an extended relational DBMS or a native XML database (NXD) system (Chaudhri et al. 2003; Liu and Murthy 2009). XQuery/XPath can be used as a query language for retrieving the data from XML documents. Even though various XML security standards (Hirsch 2002; W3C 2015) such as XML Encryption, XML Digital Signature, XML access-control markup language are defined for preserving confidentiality, integrity and access-control mechanisms of XML documents, when NXDs are used at the backend, any vulnerability in the source code of the application may allow an adversary to perform unwanted actions resulting in extraction/modification of information from/in the documents. XML injection attacks are more recent compared to SQL injection attacks, as the practice of using XML database is relatively new. A large amount of work has been done on identification of SQL injection vulnerabilities. However, only limited work has been done so far for the identification of XML injection vulnerabilities which motivated us to develop an approach for appropriate generation of attack vectors to identify XQuery injection vulnerabilities in applications driven by NXDs.

1.5 THESIS CONTRIBUTIONS

In this thesis, we investigate the challenges involved in testing web applications for identifying injection and logic vulnerabilities without using the source-code, and we develop several testing approaches to handle the identified challenges. Specifically, this thesis makes the following research contributions:

- We provide an extensive literature review on approaches available for securing web applications from injection and logic vulnerabilities. Additionally, we analyze the capabilities of existing vulnerability scanners in discovering the vulnerabilities.

- We propose a behavior-based approach for generating attack vectors through black-box analysis for the detection of logic and XQuery injection vulnerabilities in web applications.

- We model the intended behavior of the web application in terms of data flow and control flow for identifying logic vulnerabilities, and formulate an attack grammar to generate different types of XQuery attack strings for identifying all possible points of injection leading to XQuery injection attacks in the web application.

- We utilize the proposed approach to capture vulnerabilities leading to session puzzling attacks, which is not addressed in the existing literature as per our learning. In addition, we identify three new categories of XQuery injection attack vectors namely, alternate encoding, evaluation function, and XQuery comment injection attacks that are not listed in OWASP.

- We evaluate the proposed approach via extensive experiments on multiple benchmark web applications. The experimental results substantiate that the proposed approach can effectively detect the vulnerabilities existing in web applications.

1.6 THESIS ORGANIZATION

The rest of the thesis is organized as follows: Chapter 2 presents the literature review on SQL injection, XML injection, Cross-site scripting, and logic vulnerabilities/attacks. Additionally, this chapter discusses the capabilities of existing vulnerability scanners. Chapter 3 presents the problem description and objectives of this thesis. Chapters 4 and 5 elaborate the approach proposed for generation of attack vectors to detect logic and XQuery injection vulnerabilities respectively, and describe the effectiveness of the proposed approach through proper evaluation of the results. Chapter 6 summarizes the contributions of this thesis and discusses future research directions.

CHAPTER 2

LITERATURE REVIEW

Web application security has attracted much attention from both academia and industry. In order to protect web applications 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 substantial amount of research efforts has been dedicated in the past to secure web applications by preventing vulnerabilities and extenuating attacks. The discussions presented in this chapter are organized in three stages. Firstly, it provides a taxonomy for the different types of vulnerabilities in web applications and the attacks exploiting these vulnerabilities. Secondly, it presents a comprehensive review of various state-of-the-art approaches available for securing the applications from the most notorious risks like SQL injection, XML injection, cross-site scripting, session management, and logic attacks. Additionally, it highlights the promising research directions and research challenges associated with securing web applications. Finally, it discusses the capabilities of the existing vulnerability scanners, and the challenges faced by them.

2.1 WEB APPLICATION VULNERABILITIES

The attacks on web applications occur due to exploitation of vulnerabilities. The most common and widely spread vulnerabilities in web applications are: *Injection, Business Logic* and *Session Management Vulnerabilities* (Cova et al. 2007a; Felmetsger et al. 2010; Li and Xue 2013, 2014). Figure 2.1 shows the different types of vulnerabili-

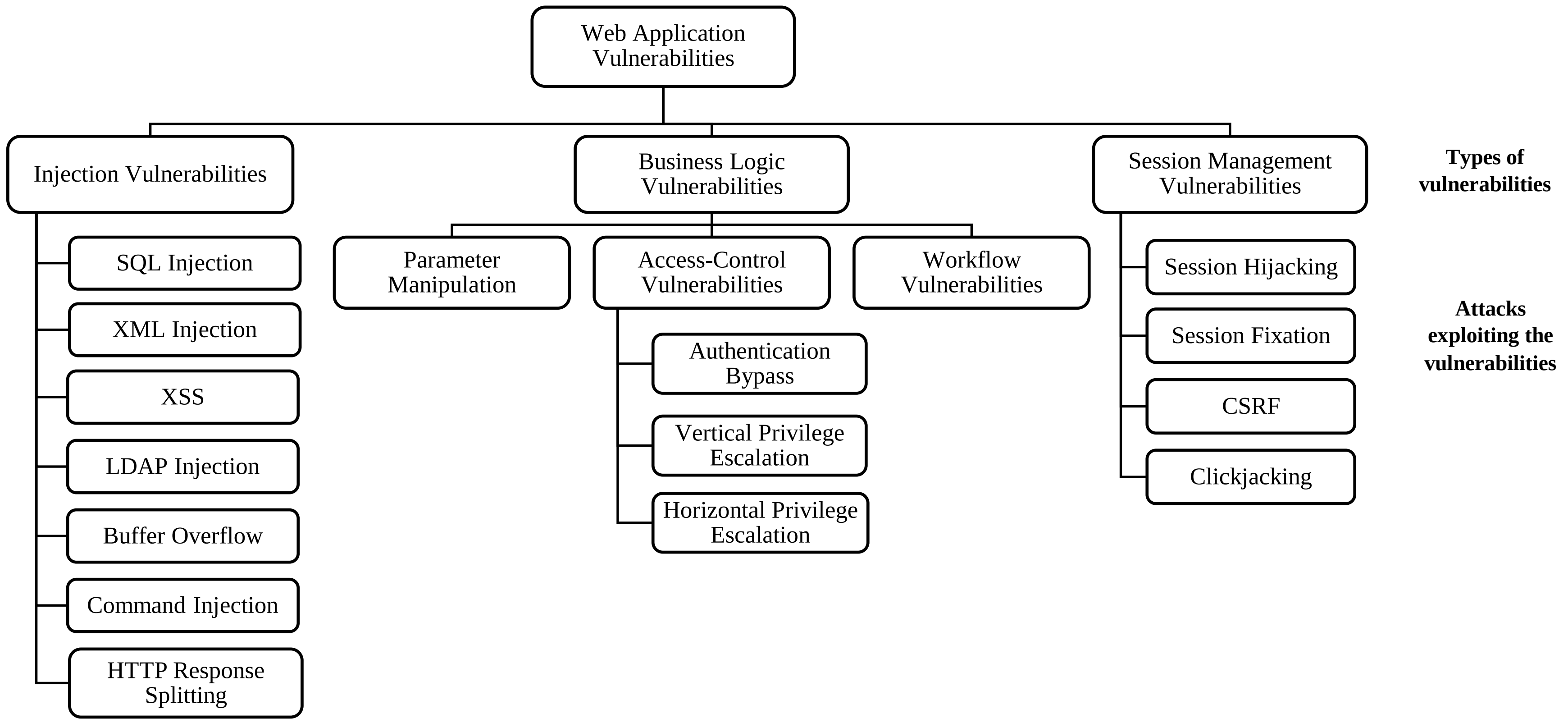
Web Application Vulnerabilities
Injection Vulnerabilities
Business Logic Vulnerabilities
Session Management Vulnerabilities
Types of vulnerabilities
SQL Injection
XML Injection
XSS
LDAP Injection
Buffer Overflow
Command Injection
HTTP Response Splitting
Parameter Manipulation
Access-Control Vulnerabilities
Workflow Vulnerabilities
Authentication Bypass
Vertical Privilege Escalation
Horizontal Privilege Escalation
Session Hijacking
Session Fixation
CSRF
Clickjacking
Attacks exploiting the vulnerabilities

Figure 2.1: Types of vulnerabilities and attacks

ties and the attacks exploiting these vulnerabilities which are discussed in detail in the following subsections.

2.1.1 Injection Vulnerabilities

Injection 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 vulnerabilities is insufficient or missing 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.

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. The 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. (2006) established a classification for SQLIAs as follows: Tautology attacks, Piggybacked queries, Union queries, Blind injection attacks, Timing attacks, Alternate encodings, Attacks on stored procedures, etc. All of the above-mentioned attacks are *First-Order SQLIAs*. There exists a special type of SQLIA called *Second-Order SQLIA* in which the malformed input is stored in the database and used at a later stage for launching the attack. Examples for these kinds of attacks can be found in Halfond et al. (2006).

XML Injection Vulnerabilities: XML injection vulnerabilities are similar to SQLIVs, and allow substitution of the malformed input in place of XPath queries/XQueries (OWASP 2015). The malformed queries on execution inject undesirable content into the XML documents and compromise the logic of an XML-based application (Stuttard and Pinto 2011). Based on the query language in which the malicious input is substituted, the injection is referred to as XPath/XQuery injection. The different types of XML injection are described in Section 5.1.

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, which leads to consequences like session hijacking, sensitive data leakage, cookie theft, and web content defacement (Stuttard and Pinto 2011). There are three types of XSS attacks, namely *Reflected, Stored* and *DOM-based XSS*. *Reflected XSS* attacks occur whenever user-input containing malicious script is referred immediately in the web page response without proper validation. *Stored XSS* attacks occur 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 the attack. These two types of attacks occur due to improper validation of user-input at the server-side. *DOM-based XSS* attacks occur at the client-side of the application (Klein 2007). These attacks make 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.

Other Injection Vulnerabilities: Command injection and LDAP injection are vulnerabilities similar to SQLIV, and substitute the malformed input in place of OS commands and LDAP statements respectively resulting in compromise of the behavior of the application (Stuttard and Pinto 2011). 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.

2.1.2 Business Logic Vulnerabilities

Business Logic Vulnerabilities are weaknesses that commonly allow attackers to manipulate the business logic of an application. They are easily exploitable, and the attacks exploiting logic vulnerabilities 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 only once, but a coding crack in the application may allow the malicious user to apply the coupon an arbitrary number of times to avail a higher percentage of discount. The most common types of logic vulnerabilities (Li and Xue 2011) are described below:

Parameter Manipulation: Manipulation of user-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 manipulation of user-input which causes violation of the semantic restrictions on the user-input. The input could be provided through the user interface, or manipulated in the HTTP request and cookies. The major reasons behind these attacks are the absence or incorrect validation 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* attacks, and the vulnerability exploited is termed as *parameter manipulation/tampering* vulnerability. For example, in an eCommerce application, failing to validate the value of credit card numbers at both client-side and server-side allows an attacker to modify the credit card number of a user in the HTTP request while placing an order, and therefore allows the attacker to purchase items for free by charging some other user.

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 users of the application to violate ACPs are called *Access-Control Vulnerabilities*. The three types of attacks that are possible due to the presence of access-control vulnerabilities are *authentication bypass*, *authorization bypass/vertical privilege escalation*, and *horizontal privilege escalation* attacks. Accessing web pages intended 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 highly privileged resource by directly pointing to the URL of a page containing the resource, then the attack is referred to as *authorization bypass/vertical privilege escalation* attack (Li and Xue 2013). If a user accesses the resource meant for another user in the same privilege level, then the attack is referred to as *horizontal privilege escalation* attack (Monshizadeh et al. 2014). According to OWASP (Top10 2013) risks and SANS errors (SANS 2011), four out of the ten risks,[1] and five out of the top 25 dangerous errors[2] are related to incorrect implementation of access-control checks.

Workflow Vulnerabilities: These vulnerabilities allow the attackers to disturb the intended workflow of the application, consequently breaking the business-specific functionality of the application. Intended workflow is the sequence of steps to be followed for completing a certain task in the application. For example, an online banking application requires the user to select a beneficiary for transferring funds, enter the amount to be transferred, and then confirm the transfer. If the "transfer amount" web page does not check whether the user has selected a beneficiary by visiting the "select beneficiary" web page only, then there is a possibility that the "select beneficiary" web page can be passed over by the user by just tampering the account number in the request to "transfer amount" page, resulting in a workflow bypass attack (Cova et al. 2007a; Li and Xue 2011; Skrupsky et al. 2013). This allows the attacker to transfer funds to their own account. This vulnerability is termed as *workflow* vulnerability, and the attack is referred to as *workflow bypass/workflow violation* attack.

[1](1) Broken Authentication and Session Management, (2) Insecure Direct Object References, (3) Missing Function Level Access-Control, and (4) Unvalidated Redirects and Forwards

[2]CWE-306, CWE-862, CWE-250, CWE-863, and CWE-732

2.1.3 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* (Braun et al. 2012; Wedman et al. 2013). *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* (Huang et al. 2012) 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 attacks target the browser to submit illegitimate requests on behalf of the user.

2.2 DEFENSIVE MECHANISMS FOR SECURING WEB APPLICATIONS

This section provides a comprehensive review of various mitigation techniques available for securing web applications from different types of attacks such as SQL/XML injection, XSS and business logic attacks. The existing literature is classified based on the approaches specified in Section 1.3.

2.2.1 Injection Defenses

This section reviews the defensive mechanisms that have been proposed for securing web applications from injection attacks such as SQL injection, XML injection, and XSS.

2.2.1.1 SQL Injection

Two major causes for SQLIAs are the ignorance towards sanitizing user-input, and framing SQL queries dynamically by concatenating SQL code and user-input using string data type during runtime. The various approaches defined in the past for addressing SQL injection 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.

(a) 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 (OWASP 2017a) 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.

(b) Vulnerability Detection: The behavior or the source code of the web application is analyzed for detecting the weaknesses existing in the application. WebSSARI, a tool developed by Huang et al. (2004), analyzes the source code of the application for extracting information flow to identify SQLIVs. While WebSSARI gathers only the intraprocedural flow of information, Xie and Aiken (2006) proposed a model that identifies inter and intra-relationship between the procedures for detecting SQLIVs. Wassermann and Su (2007) proposed a taint analysis based approach that takes into account the semantics of the validation routines for improving the effectiveness of the results.

Kosuga et al. (2007) 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 appearing in the SQL queries are identified and substituted with attack strings for identifying vulnerabilities. Sania requires the developer to generate valid HTTP requests, and employs two proxies: HTTP proxy to capture HTTP packets, and SQL proxy to capture SQL queries. Usage of two proxies is a overhead in Sania. Huang et al. (2005) implemented a framework called WAVES for detecting both SQL injection and XSS vulnerabilities in web applications. WebSSARI (Huang et al. 2004), 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. (2010) proposed a heuristic approach that infers information about tables and fields stored in the database of the application from pattern matching of the valid output and error messages obtained for legitimate and malicious test cases. The inferred information is later used for crafting attack-input that helps in identifying vulnerabilities. Medeiros et al. (2016b) developed DEKANT, a tool that learns to detect vulnerabilities automatically using machine learning. The tool operates in two phases: learning and detection. In the learning phase, a corpus is built and is used for constructing a Hidden Markov Model to train the tool about the vulnerabilities. In the detection phase, vulnerabilities are discovered using the constructed model.

Different from the previous approaches, Ceccato et al. (2016) presented a Security Oracle For Injection Attacks (SOFIA). A security oracle is a process used for evaluating the effectiveness of the test cases used for discovering vulnerabilities. The proposed oracle (i.e., SOFIA) is capable of detecting new kinds of attacks, and is not dependent on information about the test-input. Additionally, the oracle does not require access to the source code of the application under test. SOFIA operates in two phases: training and testing. During training, SQL statements executed during normal execution are clustered based on edit distance of their parse trees. During testing, the approach ascertains whether the illegitimate statement could be assigned to the clusters framed during the training. In case of negative response, a vulnerability is reported.

(c) Vulnerability Prevention: The vulnerabilities detected in an application need to be fixed before deployment for preventing the attacks. Thomas and Williams (2007) 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. (2012b) developed IPAAS, which combines machine-learning and static analysis for preventing SQL injection 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.

(d) Vulnerability Prediction: While most of the above works focus on detection of vulnerabilities, Shar and Tan (2013) developed a prototype PHPMinerI to predict SQL injection 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., SQL injection 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.

(e) Attack Detection: Lee et al. (2012) 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.). The attacks are detected during runtime after comparing the syntactic structure of the queries with predetermined one. The advantage of this approach is the capability of the algorithm in detecting the attacks at constant time. Kar et al. (2016) developed a prototype named SQLiGoT for detecting SQLIAs. SQLiGoT models the SQL queries as a graph of tokens and utilizes Support Vector Machine for classifying the queries as either genuine or malicious. Different from AMNESIA (Halfond and Orso 2005), CANDID (Bisht et al. 2010b) and SQLProb (Liu et al. 2009), SQLiGoT does not require building query models during normal execution, and can protect multiple web applications hosted on the same server.

(f) Attack Prevention: SQLGuard (Buehrer et al. 2005) prevents SQLIAs in Java web applications, when any discrepancy exists between the parse trees of the SQL statements with and without user-input. AMNESIA (Halfond and Orso 2005) 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 (Su and Wassermann 2006) constructs an augmented grammar and employs parsing technique for preventing SQLIAs. Halfond et al. (2008) 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. (2010b) 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. SQLProb (Liu et al. 2009) uses a proxy for capturing the queries which is a overhead at the server-side of the application to prevent SQLIAs. Jang and Choi (2014) 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 (2012a) 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.

Boyd and Keromytis (2004) proposed SQLRand, a randomization approach that allows developers to randomize SQL keywords for preventing SQLIAs. The randomized SQL query is de-randomized before execution using a proxy sitting between the web server and the database server. As the key used for randomization is not exposed to the attacker, the user-input with malicious code injected by the attacker is not randomized, and hence would be identified by the proxy and prevented from being executed. This approach suffers from two problems. First, using a proxy imparts an overhead on the server. Second, the approach could be circumvented if the key used for randomization of the SQL queries is exposed to the attacker. While SQLRand requires the developer to identify and randomize the SQL keywords manually, AutoRand (Perkins et al. 2016) randomizes the keywords automatically and does not require a proxy for de-randomization. AutoRand operates on bytecode of Java applications.

Different from the existing approaches, a new form of protection mechanism for databases is proposed by Medeiros et al. (2016a). The mechanism named SEPTIC (SElf-Protecting daTabases preventIng attaCks) embeds the protection module within the database to either detect or prevent SQL injection and stored injection attacks such as second-order SQL injection, stored XSS, etc. against the database. Table 2.1 summarizes the literature on SQL injection.

2.2.1.2 XML Injection

This section provides insight about the work done so far for identifying/preventing injection vulnerabilities/attacks that target the XML documents involved in the web application. With respect to SQL injection, relatively less amount of research has been done on XML injection vulnerability detection and attack prevention. A detailed classification of the existing approaches on SQL/XML injection can be found in the literature (Deepa and Thilagam 2016; Halfond et al. 2006; Li and Xue 2014; Palsetia et al. 2016; Shahriar and Zulkernine 2011b, 2012b). Most of the research works addressing XML injection are inclined to prevention of attacks and only a few works exist for identifying vulnerabilities.

Table 2.1: Summary of SQL injection prevention/detection approaches

	Area of Focus					**Type of Analysis**					**Approach**				
Research Article	Vulnerability Detection	Vulnerability Prevention	Attack Detection	Attack Prevention	Vulnerability Prediction	Secure Programming	Static Analysis	Dynamic Analysis	Penetration Testing	Others	Signature-Based	Query Model-Based	Knowledge-Based	Machine Learning	Instruction-set Randomization
WebSSARI (Huang et al. 2004)	✓						✓								
SQLRand (Boyd and Keromytis 2004)				✓						✓					✓
SQLGuard (Buehrer et al. 2005)				✓			✓					✓			
AMNESIA (Halfond and Orso 2005)			✓	✓			✓	✓				✓			
Waves (Huang et al. 2005)	✓							✓	✓				✓		
SQLCheck (Su and Wassermann 2006)				✓			✓					✓			
Xie and Aiken (2006)	✓						✓								
Sania (Kosuga et al. 2007)	✓							✓	✓			✓			
Thomas and Williams (2007)		✓				✓	✓				✓				
Wassermann and Su (2007)	✓						✓								
WASP (Halfond et al. 2008)				✓				✓							
SQLProb (Liu et al. 2009)				✓				✓				✓			
CANDID (Bisht et al. 2010b)				✓				✓				✓			

Continued on next page

Table 2.1 – Summary of SQL injection prevention/detection approaches (continued from previous page)

	Area of Focus					Type of Analysis					Approach				
Research Article	Vulnerability Detection	Vulnerability Prevention	Attack Detection	Attack Prevention	Vulnerability Prediction	Secure Programming	Static Analysis	Dynamic Analysis	Penetration Testing	Others	Signature-Based	Query Model-Based	Knowledge-Based	Machine Learning	Instruction-set Randomization
Ciampa et al. (2010)	✓							✓							
IPAAS (Scholte et al. 2012b)		✓					✓							✓	
Lee et al. (2012)			✓				✓	✓							
PHPMinerI (Shar and Tan 2013)					✓		✓			✓				✓	
Jang and Choi (2014)				✓				✓							
DEKANT (Medeiros et al. 2016b)	✓						✓							✓	
SQLiGoT (Kar et al. 2016)			✓							✓				✓	
AutoRand (Perkins et al. 2016)				✓						✓					✓
SEPTIC (Medeiros et al. 2016a)			✓	✓				✓				✓			
SOFIA (Ceccato et al. 2016)	✓							✓	✓			✓		✓	

(a) Secure Programming: Secure coding practices to prevent injection attacks include validation of user-supplied input, parameterizing queries, pre-compilation of queries, escaping the special characters in the input, and embedding grammar of the query language into that of the host language (e.g., Java, C#, VB, etc.) (Bravenboer et al. 2010; Truelove and Svoboda 2011; XPath-Injection 2015). The drawback with this approach is the overhead imparted on developers to learn and implement the security paradigms during construction.

(b) Vulnerability Detection: Antunes et al. (2009) identified injection vulnerabilities in web services using Aspect Oriented Programming that intercept all the calls to API methods executing SQL commands. The structure of SQL/XPath commands issued in the presence of attacks to the ones previously learnt when running the workload in the absence of attacks are compared for identifying the vulnerabilities. Sign-WS (Antunes and Vieira 2011) employs penetration testing and interface monitoring for detection of the attack signatures to identify injection vulnerabilities in web services. Laranjeiro et al. (2009) proposed a query-model based approach that combines penetration testing and static code analysis to detect SQL/XPath vulnerabilities and therefore prevent SQL and XPath injection attacks in web services. Jan et al. (2016) developed a testing framework (SOLMI) for identifying XML injection vulnerabilities in web services. It works based on mutation operators defined to manipulate XML messages and a constraint solver for generation of test cases to provide valid but malicious messages. The constraint solver generates nested attacks to increase the effectiveness. Jan et al. (2017) developed a search-based testing approach, an extension of SOLMI, for identifying XML injection vulnerabilities in web services. The approach is automated, and it utilizes genetic algorithm for identifying test cases that are effective in discovering the vulnerabilities. While SOLMI defines mutation operators for manipulating XML messages, this approach concentrates on effective generation of test-inputs that are capable of delivering malicious XML messages.

(c) Attack Prevention: Huang (2003) discussed detection of intrusions in database systems through fingerprinting transactions. A broad class of injection attacks is prevented using a signature-based approach in Mitropoulos et al. (2011). During training phase, vulnerable code statements are identified and registered using unique signatures for differentiating normal and abnormal executions. During runtime, the framework checks the compliance of all statements with the learnt model and blocks code statements containing maliciously injected elements. New unknown attacks cannot be detected using the signature-based approach, even if they have only small variations from a known payload (Rosa et al. 2013). Groppe and Groppe (2008) identified XPath queries that do not satisfy the constraints defined in the schema, and forbid the queries from being executed for preventing attacks. Lampesberger (2013) detected anomalies in XML documents from grammatical-inference of the documents by constructing a visibly pushdown automaton. The automaton provides the syntactic structure and data types of the parameters used in the XML document, which helps in identifying the anomalies. Asmawi et al. (2012) proposed a query-model based approach similar to Antunes et al. (2009) for preventing XPath injection attacks in a web services environment. The problem with query model-based approach is its failure in detecting attacks where the structure of the query varies dynamically based on conditional input (Bisht et al. 2010b).

Mitropoulos et al. (2009) developed a methodology that constructs a knowledge-base during the training phase. During the testing phase, when an XPath query is encountered, an identifier is generated. If the identifier does not exist in the knowledge-base, then the query is considered as malicious. The major drawback of this technique is that any change in source code requires a new training to reassign the identifiers. Rosa et al. (2013) presented an XML injection strategy-based detection system for mitigating zero-day attacks. A machine learning based approach proposed by Valeur et al. (2005) calculated anomaly scores for each query being executed, and detected SQLIAs when the anomaly score exceeds the maximum anomaly score obtained during attack free execution. Anomalies in XML transactions are detected using a framework called XML-AD (Menahem et al. 2012). The features of the XML documents are extracted and transformed into attribute vectors, and anomalies are detected using a novel multi-univariate anomaly detection algorithm, ADIFA. A predictive fuzzy associative rule model (FARM) (Chan et al. 2013) is developed for identifying attack patterns and anomalies to counter both signature and anomaly-based XML related attacks. The problem with the machine learning approach is that the number of false positives and false negatives generated depends on appropriate selection of the training set. The existing approaches focusing on XML injection are summarized in Table 2.2.

2.2.1.3 Cross-Site Scripting

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, 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.

(a) Secure Programming: XSS vulnerabilities can be eliminated by adopting secure coding practices like sanitization of untrusted input for removing 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 (OWASP 2017b). The implementation overhead imparted on developers for writing secure code can be addressed in Java applications by using the library Stones (Juillerat 2007) that prevents SQL injection 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 developers. 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. (2010), 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. (2012) to enforce secure programming guidelines for preventing XSS attacks.

Table 2.2: Summary of XML injection prevention/detection approaches

	Area of Focus				Type of Analysis				Approach						
Research Article	Vulnerability Detection	Vulnerability Prevention	Attack Detection	Attack Prevention	Static Analysis	Dynamic Analysis	Penetration Testing	Others	Signature-Based	Schema-Based	Query Model-Based	Knowledge-Based	Machine Learning	Mutation-Based	Search-Based
Huang (2003)				✓					✓						
Groppe and Groppe (2008)				✓						✓					
Antunes et al. (2009)	✓					✓	✓				✓				
Laranjeiro et al. (2009)	✓				✓		✓				✓				
Mitropoulos et al. (2009)				✓	✓							✓			
Antunes and Vieira (2011)	✓						✓		✓						
Mitropoulos et al. (2011)				✓					✓						
Asmawi et al. (2012)				✓	✓						✓				
Menahem et al. (2012)				✓									✓		
Chan et al. (2013)				✓									✓		
Lampesberger (2013)				✓						✓					
Rosa et al. (2013)				✓	✓				✓			✓			
Jan et al. (2016)	✓							✓						✓	
Jan et al. (2017)	✓						✓								✓

(b) Vulnerability Detection: Di Lucca et al. (2004) 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 (Scholte et al. 2012a; Wassermann and Su 2008). *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 the untrusted input is used in the HTML output statements (i.e., sensitive sinks) without encountering sanitization. Jovanovic et al. (2006b,c) 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 (2008) and Balzarotti et al. (2008). Wassermann and Su (2008) 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. (2008) 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. DEKANT (Medeiros et al. 2016b) discussed in Section 2.2.1.1 detects XSS vulnerabilities in web applications using machine learning approach.

Kals et al. (2006) developed Secubat, a black-box vulnerability scanner, for identifying SQL and XSS vulnerabilities. Secubat uses a crawler to identify the web pages in the application, fills the form fields with attack vectors, and then analyzes the output for detecting vulnerabilities. It is capable of detecting only reflected XSS. A black-box fuzzer, KameleonFuzz, developed by Duchene et al. (2014) automates the generation of malicious inputs using genetic algorithm for detecting XSS vulnerabilities. While, the previous works (Balzarotti et al. 2008; Di Lucca et al. 2004; Duchene et al. 2014; Kals et al. 2006; Wassermann and Su 2008) concentrate on detecting vulnerabilities by injecting malicious input, Shahriar and Zulkernine (2009) took the first step towards injecting faults into the source code for generating sufficient number of test-cases to detect XSS vulnerabilities in PHP applications.

Shar and Tan (2012) 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. (2012) 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) (Brijesh Deb and Bharti 2007; Ward 2007). RIAs are the applications that are rich in content (i.e., audio, video, etc.), highly interactive and responsive, and involve technologies like Flash, Silverlight, Asynchronous JavaScript and XML (AJAX), etc. for rendering web pages. While most of the aforementioned approaches cannot detect DOM-based XSS vulnerabilities, Lekies et al. (2013) proposed an approach to detect DOM-based XSS using a taint-aware JavaScript engine.

(c) Vulnerability Prevention: IPAAS (Scholte et al. 2012b), described in Section 2.2.1.1, enforces validation policies on input parameters during runtime to prevent XSS vulnerabilities. IPAAS is different from the approaches by Balzarotti et al. (2008); Jovanovic et al. (2006b,c); Wassermann and Su (2008) as it enforces 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 hence cannot assure complete enforcement of security constraints. ScriptGard developed by Saxena et al. (2011) detects erroneous placement of sanitization routines and repairs them. Doupé et al. (2013) 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. (2010) is similar to deDacota, and it enables secure construction of new applications, while deDacota ensures security of legacy applications.

(d) Vulnerability Prediction: As already described in Section 2.2.1.1, PHPMinerI (Shar and Tan 2013) predicts the vulnerabilities based on attributes defined for reflecting properties of sanitization routines. Figure 2.2 summarizes the list of articles focusing on detection and prevention of XSS vulnerabilities.

(e) Attack Detection: XSSDS (Johns et al. 2008) 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 request have become a part of the client-side script in the response page. Shahriar and Zulkernine (2011a) developed a framework 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. (2009) 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 et al. (2014a,b) 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 the work by Shahriar et al. (2014b).

(f) Attack Prevention: XSS-Guard (Bisht and Venkatakrishnan 2008) works in a similar way to XSSDS (Johns et al. 2008), and aims at identifying and removing malicious scripts unintended by the web application. Blueprint (Ter Louw and Venkatakrishnan 2009), an extension of XSS-Guard, prevents execution of unauthorized scripts by

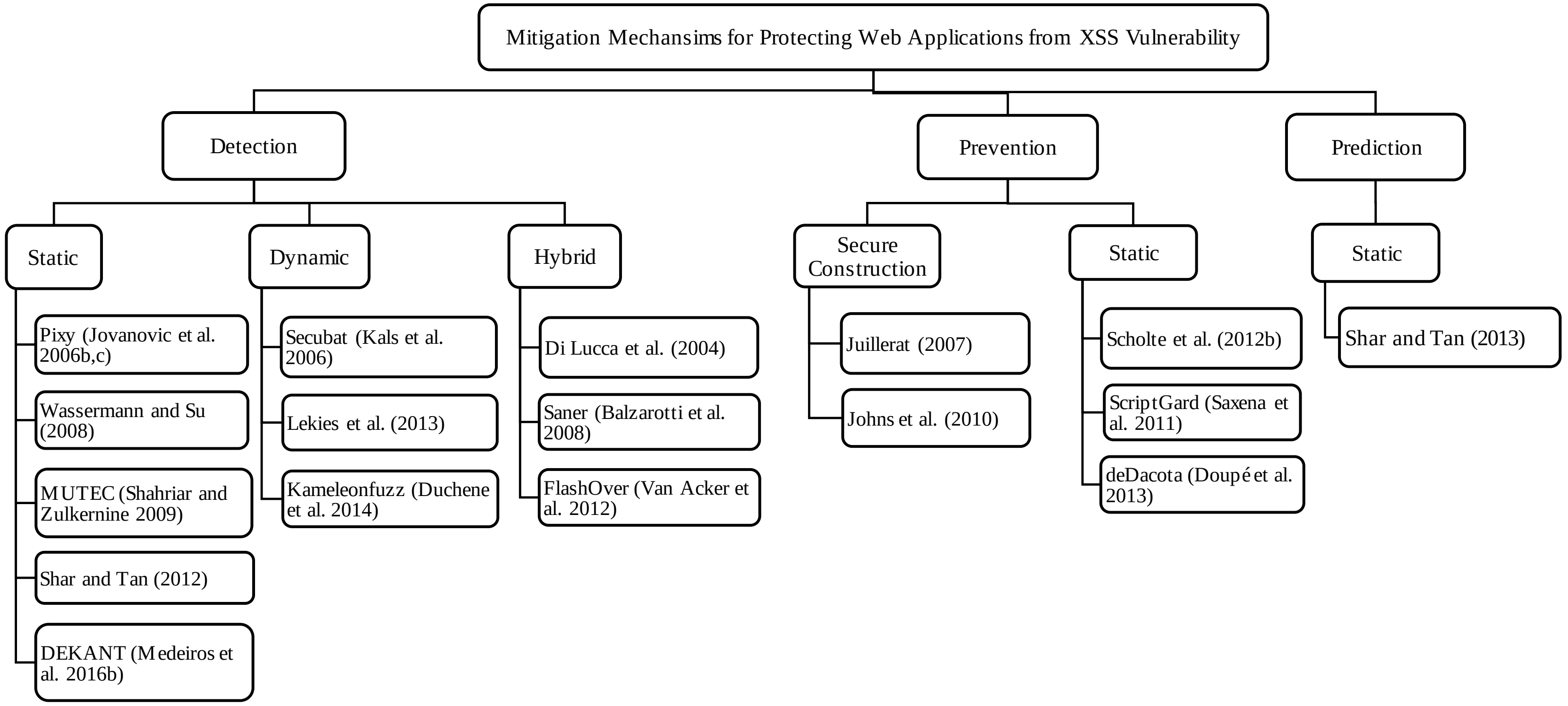
Mitigation Mechansims for Protecting Web Applications from XSS Vulnerability
Detection
Prevention
Prediction
Static
Dynamic
Hybrid
Secure Construction
Static
Static
Pixy (Jovanovic et al. 2006b,c)
Wassermann and Su (2008)
MUTEC (Shahriar and Zulkernine 2009)
Shar and Tan (2012)
DEKANT (Medeiros et al. 2016b)
Secubat (Kals et al. 2006)
Lekies et al. (2013)
Kameleonfuzz (Duchene et al. 2014)
Di Lucca et al. (2004)
Saner (Balzarotti et al. 2008)
FlashOver (Van Acker et al. 2012)
Juillerat (2007)
Johns et al. (2010)
Scholte et al. (2012b)
ScriptGard (Saxena et al. 2011)
deDacota (Doupé et al. 2013)
Shar and Tan (2013)

Figure 2.2: Research on detection and prevention of XSS vulnerabilities

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 Blueprint is the requirement of programmer to annotate statements holding untrusted contents. Noncespaces (Van Gundy and Chen 2009, 2012) enables the clients to differentiate malicious from non-malicious contents 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 (Cao et al. 2011) prevents all types of XSS attacks by preventing propagation of malicious scripts in social networking websites.

Chaudhuri and Foster (2010) developed a framework Rubyx for detecting XSS, CSRF and session manipulation vulnerabilities in Ruby-on-Rails web applications. ScriptGard developed by Saxena et al. (2011) detects the faults in sanitization routines and removes them. SessionSafe (Johns 2006) 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 SQL injection and XSS vulnerabilities is analyzed using a Fuzzy Logic system (Shahriar and Haddad 2014). SEPTIC (Medeiros et al. 2016a) protects the web applications from stored XSS attacks arising due to malicious scripts stored in the database of the application.

While all of the above-mentioned works (Balzarotti et al. 2008; Bisht and Venkatakrishnan 2008; Di Lucca et al. 2004; Doupé et al. 2013; Duchene et al. 2014; Johns et al. 2008; Jovanovic et al. 2006b,c; Kals et al. 2006; Saxena et al. 2011; Scholte et al. 2012b; Shahriar and Zulkernine 2011a; Shar and Tan 2012; Ter Louw and Venkatakrishnan 2009; Van Acker et al. 2012; Wassermann and Su 2008; Wurzinger et al. 2009) deploy security mechanisms at the server-side of the application, BEEP (Jim et al. 2007), Noxes (Kirda et al. 2009, 2006), Noncespaces (Van Gundy and Chen 2009, 2012), Vogt et al. (2007), and Stock et al. (2014) deploy the security mechanisms at the client-side of the application for mitigating XSS attacks.

BEEP (Jim et al. 2007) enhances the browsers with policies that specify the scripts to be executed from a web page. Noxes (Kirda et al. 2009, 2006) 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. (2007) tracks the flow of sensitive information inside the web browser for protecting the application against XSS attack. Both Noxes and Vogt et al. (2007) 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. (2014) extended the work by Lekies et al. (2013), and combined the taint-aware JavaScript engine with taint-aware parsers for preventing DOM-based XSS attacks. Figure 2.3 summarizes the list of articles focusing on detection and prevention of XSS attacks.

The detailed review on Reflected-XSS and Stored-XSS attacks (Garcia-Alfaro and Navarro-Arribas 2008, 2009) states that XSS can be prevented by either employing policies for filtering malicious code or enforcing security policies at the browser-end. Table 2.3 provides a summary of the literature, and depicts that most of the existing works focus on detection of XSS vulnerabilities and prevention of XSS attacks. ScriptGard (Saxena et al. 2011) and the work by Shar and Tan (2012) are the only works concentrating on the removal of XSS vulnerabilities. Also, only a little amount of work is done towards preventing DOM-based XSS attack (Klein 2007), a third kind of XSS attack emerging due to implementation of business logic at the client-side of the application.

2.2.1.4 Research Challenges and Directions

SQL Injection: The existing solutions for detecting SQL injection have their own pros and cons, and they are described as follows. Secure programming imparts overhead on developers for implementing the security guidelines during development (Bravenboer et al. 2010). Signature-based approach does not prevent zero-day attacks, and suffers from false negatives when attack query matches the structure of a legitimate query (Thomas and Williams 2007). The knowledge-based approach requires new

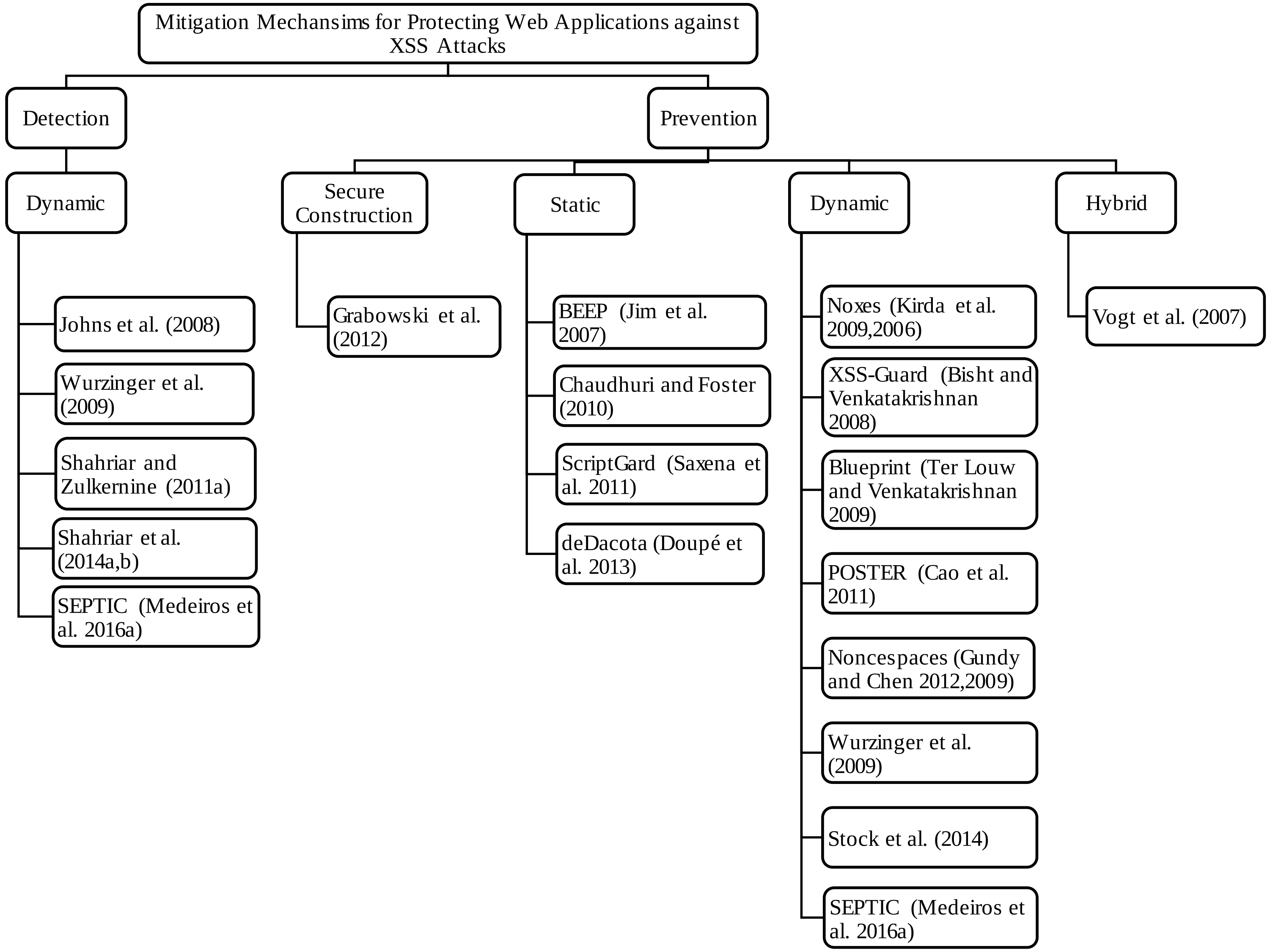
Mitigation Mechansims for Protecting Web Applications against XSS Attacks
Detection
Prevention
Dynamic
Johns et al. (2008)
Wurzinger et al. (2009)
Shahriar and Zulkernine (2011a)
Shahriar et al. (2014a,b)
SEPTIC (Medeiros et al. 2016a)
Secure Construction
Grabowski et al. (2012)
Static
BEEP (Jim et al. 2007)
Chaudhuri and Foster (2010)
ScriptGard (Saxena et al. 2011)
deDacota (Doupé et al. 2013)
Dynamic
Noxes (Kirda et al. 2009,2006)
XSS-Guard (Bisht and Venkatakrishnan 2008)
Blueprint (Ter Louw and Venkatakrishnan 2009)
POSTER (Cao et al. 2011)
Noncespaces (Gundy and Chen 2012,2009)
Wurzinger et al. (2009)
Stock et al. (2014)
SEPTIC (Medeiros et al. 2016a)
Hybrid
Vogt et al. (2007)

Figure 2.3: Research on detection and prevention of XSS attacks

Table 2.3: Summary of articles on XSS

		Area of Focus					**Type of Analysis**			**Type of XSS**		
Research Article	Year	Vulnerability Detection	Vulnerability Prevention	Attack Detection	Attack Prevention	Vulnerability Prediction	Secure Programming	Static Analysis	Dynamic Analysis	Reflected XSS	Stored XSS	DOM-Based XSS
Di Lucca et al. (2004)	2004	✓						✓	✓	•	•	•
Pixy (Jovanovic et al. 2006b,c)	2006	✓						✓		•	•	•
Wassermann and Su (2008)	2008	✓						✓		✓	✓	
MUTEC (Shahriar and Zulkernine 2009)	2009	✓						✓		✓	✓	✓
Shar and Tan (2012)	2012	✓						✓		✓	✓	
DEKANT (Medeiros et al. 2016b)	2016	✓						✓		✓	•	•
Secubat (Kals et al. 2006)	2006	✓							✓	✓		
Saner (Balzarotti et al. 2008)	2008	✓						✓	✓	•	•	•
FlashOver (Van Acker et al. 2012)	2012	✓						✓	✓	•	•	•
Lekies et al. (2013)	2013	✓							✓			✓
Kameleonfuzz (Duchene et al. 2014)	2014	✓							✓	✓	✓	
Grabowski et al. (2012)	2012				✓		✓			•	•	•
BEEP (Jim et al. 2007)	2007				✓			✓		✓	✓	
Chaudhuri and Foster (2010)	2010				✓			✓		•	•	•
ScriptGard (Saxena et al. 2011)	2011		✓		✓			✓		•	•	•
Vogt et al. (2007)	2007				✓			✓	✓	✓	✓	✓
XSS-Guard (Bisht and Venkatakrishnan 2008)	2008				✓				✓	•	•	•
Noxes (Kirda et al. 2009, 2006)	2009				✓				✓	✓	✓	
Blueprint (Ter Louw and Venkatakrishnan 2009)	2009				✓				✓	✓	✓	

Continued on next page

Table 2.3 – Summary of articles on XSS (continued from previous page)

		Area of Focus					**Type of Analysis**			**Type of XSS**		
Research Article	Year	Vulnerability Detection	Vulnerability Prevention	Attack Detection	Attack Prevention	Vulnerability Prediction	Secure Programming	Static Analysis	Dynamic Analysis	Reflected XSS	Stored XSS	DOM-Based XSS
POSTER (Cao et al. 2011)	2011				✓				✓	✓	✓	✓
Noncespaces (Van Gundy and Chen 2009, 2012)	2012				✓				✓	✓	✓	
Stock et al. (2014)	2014				✓				✓			✓
Juillerat (2007)	2007		✓				✓			•	•	•
Johns et al. (2010)	2010		✓				✓			•	•	•
Scholte et al. (2012b)	2012		✓					✓		•	•	•
deDacota (Doupé et al. 2013)	2013		✓		✓			✓		✓	✓	
Johns et al. (2008)	2008			✓					✓	✓	✓	
Wurzinger et al. (2009)	2009			✓	✓				✓	•	•	•
Shahriar and Zulkernine (2011a)	2011			✓					✓	•	•	•
Shahriar et al. (2014a,b)	2014			✓					✓	•	•	•
SEPTIC (Medeiros et al. 2016a)	2016			✓	✓				✓		✓	
Shar and Tan (2013)	2013					✓		✓		•	•	•

• - Not specified in the paper.

training whenever modifications are made to the source code of the application (Huang et al. 2005). The accuracy of machine learning approach is dependent on appropriate selection of the training set (Ceccato et al. 2016; Kar et al. 2016; Medeiros et al. 2016b; Scholte et al. 2012b; Shar and Tan 2013).

Even though various mechanisms exist for securing web applications from SQL injection, most of the proposed solutions fail to address all kinds of SQLIAs. Existing methods proposed by Bisht et al. (2010b), Halfond and Orso (2005), Buehrer et al. (2005), and Su and Wassermann (2006) cannot protect against SQLIAs on stored procedures. The existing works focus on detection/prevention of known patterns of SQLIAs, and hence are unable to identify zero-day attacks. Most of the works address first-order SQLIAs only, and very little attention is paid towards addressing second-order SQLIAs.

XML Injection: Existing approaches for addressing XML injection concentrate on detection/prevention of vulnerabilities/attacks in web services only, and cover a certain type of XML injection attacks only. The existing open-source penetration testing tools for identification of XML injection vulnerabilities cover only certain types of XML injection attacks. XCat (Forbes 2014) is a tool used to identify blind XPath injection vulnerabilities in web applications. WebCruiser (2011) is designed to detect XML injection, but it detects only tautology injection attacks and cannot detect other types of XML injection attacks. The vulnerability scanners such as Acunetix, Wapiti,[3] W3af,[4] and Arachni (2016) identify XPath injection vulnerabilities in web applications involving XML documents (van der Loo 2011). Each of the existing tools covers only a specific type of vulnerability, such as tautology attacks, or focuses on XPath injection. In addition, none of the existing tools detect XML vulnerabilities in web applications using NXDs. Therefore, there is a need for an approach that focuses on detection of various types of XQuery injection vulnerabilities in web applications using NXDs. Hence, this work focuses on development of an approach to generate attack vectors for the detection of XQuery injection vulnerabilities in web applications driven by NXDs.

[3]http://wapiti.sourceforge.net/
[4]http://w3af.sourceforge.net/

Cross-site Scripting: In spite of the large amount of effort 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. (2011) 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. Therefore, the development of frameworks that are sensitive to the context of the applications is an interesting area of research. Additionally, the research works addressing DOM-based XSS are a very few, and hence addressing it would be a promising research direction.

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

2.2.2 Session Management Defenses

Session management is necessary 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 (Wedman et al. 2013). Therefore the secure coding practices for preventing session management attacks involve the generation of session tokens using long random numbers, the generation of a new token whenever a user logs-in to the application rather than elevation of an already created token (Johns et al. 2011), and deployment of proper timeout mechanisms for destroying the session tokens. In addition to SMVs, XSS vulnerabilities also pave way for session management attacks (Evans and Shahriar 2014). Therefore, proper defense mechanisms employed for preventing XSS vulnerabilities in turn avoid session management attacks as well. SessionSafe (Johns 2006) aims at preventing session hijacking attacks resulting from XSS vulnerabilities.

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 (Jovanovic et al. 2006a). 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. (2008) 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. (2012) 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 (Huang et al. 2012; Johns and Lekies 2013; Shahriar and Devendran 2014), and so on. Shahriar et al. (2013) proposed ProClick, a proxy-level framework for identifying clickjacking attacks. Braun et al. (2012) 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.

2.2.3 Business Logic Defenses

This subsection deals with various defensive approaches available for securing web applications from business logic attacks. The articles are categorized based on the type of logic attack addressed and the type of security mechanism employed for mitigating the attack.

2.2.3.1 Parameter Tampering

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.

(a) Secure Construction of New Web Applications: Swift (Chong et al. 2007a,b) is a programming model built on the top of Jif language to enable secure construction of web applications. Swift ensures confidentiality 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 (Vikram et al. 2009), a successor of Swift (Chong et al. 2007a), 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 (Yip et al. 2009) 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 SQL injection, XSS, and missing access-control checks.

The frameworks Swift (Chong et al. 2007a,b), Ripley (Vikram et al. 2009), and Resin (Yip et al. 2009) 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.

(b) Vulnerability Detection: 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. (2010a, 2011) formulated a systematic approach towards detection of parameter tampering vulnerabilities by devising two tools, NoTamper (Bisht et al. 2010a) and WAPTEC (Bisht et al. 2011). 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. The vulnerabilities are detected by observing the response obtained for user-input violating the checks imposed on them. NoTamper 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. Additionally, it does not take into account the data flow within the application. WAPTEC 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 web application 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. (2012) 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.

WAPTEC and ViewPoints analyze the code bases at different layers of the web application architecture and identify inconsistencies existing in each layer. 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 after analyzing the source code, Balduzzi et al. (2011) designed an automated black-box 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).

(c) Attack Prevention: Mouelhi et al. (2011) defined a black-box approach for protecting the web applications 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 the 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 as follows: 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 (Skrupsky et al. 2013) 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 CSRF prevention (Jovanovic et al. 2006a), 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). Figure 2.4 provides a summary of research articles addressing parameter manipulation vulnerabilities.

2.2.3.2 Access-Control Violation

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 prevention of access-control violations are described in the following subsections.

(a) Secure Construction 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. (2008) developed an intermediate programming language Aura that acts as a typechecker and interpreter for supporting authorization policies of web application. Swamy et al. (2008) 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. (2009) developed a programming language SELinks integrating the type system Fable

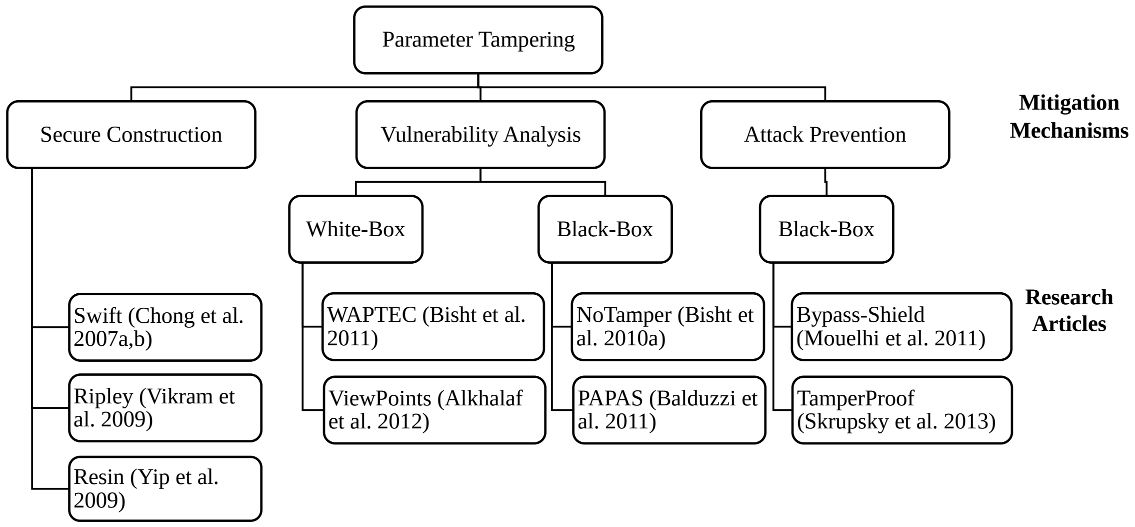
Parameter Tampering
Secure Construction
Vulnerability Analysis
Attack Prevention
Mitigation Mechanisms
White-Box
Black-Box
Black-Box
Research Articles
Swift (Chong et al. 2007a,b)
Ripley (Vikram et al. 2009)
Resin (Yip et al. 2009)
WAPTEC (Bisht et al. 2011)
ViewPoints (Alkhalaf et al. 2012)
NoTamper (Bisht et al. 2010a)
PAPAS (Balduzzi et al. 2011)
Bypass-Shield (Mouelhi et al. 2011)
TamperProof (Skrupsky et al. 2013)

Figure 2.4: Research on parameter manipulation vulnerabilities

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 (Swamy et al. 2010) 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 (Morgenstern and Licata 2010) is a library extending the features of Aura, FINE, 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. (2010) developed Capsules, a framework built on the top of Java Servlet enabling limited access to highly secured resources and therefore minimizes the impairment to resources due to attackers.

(b) Vulnerability Detection: Waler (Felmetsger et al. 2010), an extension of MiMoSA (Balzarotti et al. 2007), 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 (Cova et al. 2007a) 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 (Felmetsger et al. 2010) is able to identify a wide range of logic vulnerabilities in web applications, Sun et al. (2011) suggested a static analysis approach to detect access-control vulnerabilities. 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 access-control vulnerabilities. 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 (Felmetsger et al. 2010), which was dependent on the invariants inferred during execution of the application.

Son et al. (2011) developed a prototype RoleCast, which is used for identifying access-control vulnerabilities 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. Son and Shmatikov (2011) developed another tool called SaferPHP that employs taint analysis for detecting missing access checks in PHP applications.

FixMeUp (Son et al. 2013), a follow-up work of RoleCast (Son et al. 2011), is the first tool to eliminate access-control vulnerabilities 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. Monshizadeh et al. (2016) developed a prototype named LogicPatcher that identifies missing and incomplete access checks existing in applications with minimal guidelines as compared to FixMeUp, and patches them with fixes.

MiMoSA (Balzarotti et al. 2007) and Waler (Felmetsger et al. 2010) extract a model out of the source code and then use a model checker to detect any violation of invariants, whereas RoleCast, FixMeUp, and the prototype by Sun et al. (2011) 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 (Dalton et al. 2009; Felmetsger et al. 2010; Son et al. 2011; Sun et al. 2011) focus only on detection of vulnerabilities, FixMeUp and LogicPatcher patches these vulnerabilities with fixes.

LogicScope (Li and Xue 2013) models the business logic of the web application using a finite state machine, and the discrepancies between the intended and implemented finite state machines 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 during construction of the input symbols.

Li et al. (2014) implemented another prototype system BATMAN for disclosing access-control vulnerabilities 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. Wen et al. (2016) developed Scout for identifying access-control vulnerabilities in applications which use MongoDB at the backend. Scout infers the intended ACPs from the execution traces and develops a access operation model based on the access-control operations with the MongoDB. Later, the application is exploited based on the policies inferred from the model for discovery of vulnerabilities. LogicScope, BATMAN, and Scout employ black-box approach for detecting the vulnerabilities.

While the above-mentioned works (Li et al. 2014; Li and Xue 2011; Li et al. 2012) 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. (2014) proposed a framework called MACE that analyzes the source code 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. Zhu et al. (2015) developed a plug-in called ASIDE for the IDEs (Integrated Development Environment) such as Eclipse to support PHP applications. The plug-in prompts the developer to indicate security-sensitive components in the code, both to remind them to incorporate access checks during construction and to annotate application-specific ACPs, and therefore supports secure programming. The annotations are used during analysis of the source code for discovering access-control vulnerabilities. The analysis is run as a background process in the IDE, and is termed as interactive static analysis, as it is performed with the help of annotations specified by the developer during construction of the application within the IDE.

(c) Attack Prevention: Li and Xue (2011) 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 (Cova et al. 2007a), 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 (Li et al. 2012), a follow-up work of BLOCK, 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 the performance overhead imposed due to interception of each SQL query for evaluation.

Parno et al. (2009) 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. (2009), prevents authentication and authorization attacks in legacy web applications. It combines authentication information of a user with developer-provided access-control lists 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. Muthukumaran et al. (2015) developed FlowWatcher, a proxy-based solution preventing leakage of sensitive data to anonymous and unprivileged users of the application by monitoring the HTTP traffic. FlowWatcher operates based on the user-data-access policy specified by the application developer, which reflects the intended ACP of the application. Figure 2.5 provides a summary of research articles addressing access-control vulnerabilities.

2.2.3.3 Workflow Violation

Balzarotti et al. (2007) developed a prototype Multi-Module State Analyzer (MiMoSA) to identify workflow violation attacks in PHP applications. While some of the aforementioned works (Dalton et al. 2009; Li et al. 2014; Li and Xue 2011; Li et al. 2012;

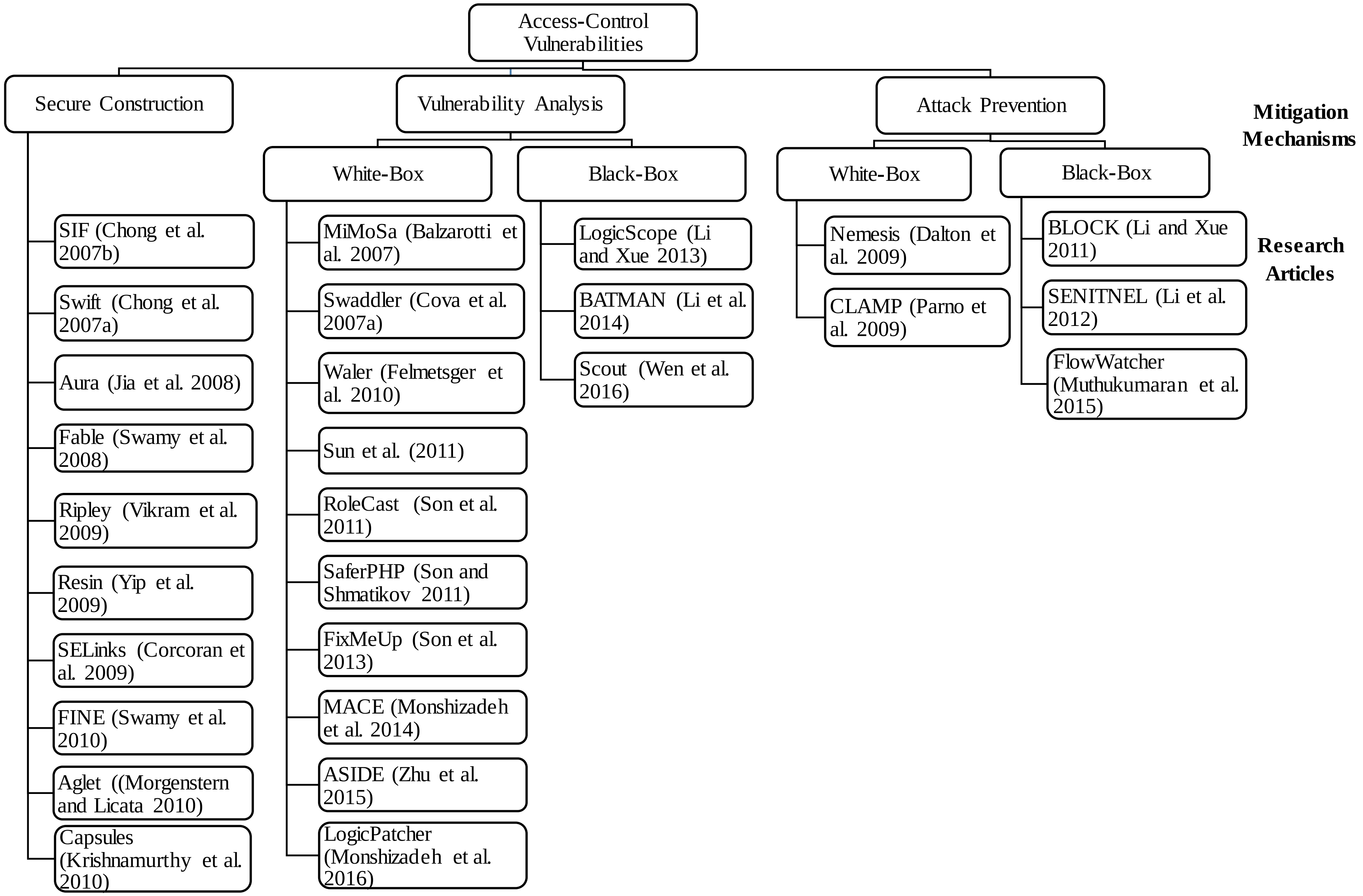
Access-Control Vulnerabilities
Secure Construction
Vulnerability Analysis
Attack Prevention
Mitigation Mechanisms
White-Box
Black-Box
White-Box
Black-Box
Research Articles
SIF (Chong et al. 2007b)
Swift (Chong et al. 2007a)
Aura (Jia et al. 2008)
Fable (Swamy et al. 2008)
Ripley (Vikram et al. 2009)
Resin (Yip et al. 2009)
SELinks (Corcoran et al. 2009)
FINE (Swamy et al. 2010)
Aglet ((Morgenstern and Licata 2010)
Capsules (Krishnamurthy et al. 2010)
MiMoSa (Balzarotti et al. 2007)
Swaddler (Cova et al. 2007a)
Waler (Felmetsger et al. 2010)
Sun et al. (2011)
RoleCast (Son et al. 2011)
SaferPHP (Son and Shmatikov 2011)
FixMeUp (Son et al. 2013)
MACE (Monshizadeh et al. 2014)
ASIDE (Zhu et al. 2015)
LogicPatcher (Monshizadeh et al. 2016)
LogicScope (Li and Xue 2013)
BATMAN (Li et al. 2014)
Scout (Wen et al. 2016)
Nemesis (Dalton et al. 2009)
CLAMP (Parno et al. 2009)
BLOCK (Li and Xue 2011)
SENITNEL (Li et al. 2012)
FlowWatcher (Muthukumaran et al. 2015)

Figure 2.5: Research on access-control vulnerabilities

Monshizadeh et al. 2014; Parno et al. 2009) focus on identifying insecure data flow, MiMoSA utilizes the source code to characterize the intended workflow of the application and variables (e.g., session variables, cookies, etc.) that aid in maintaining the state of the application.

Swaddler (Cova et al. 2007a) 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 (Felmetsger et al. 2010), a follow-up work of Swaddler (Cova et al. 2007a) is able to identify workflow violations in addition to access-control vulnerabilities.

LogicScope (Li and Xue 2013) can detect workflow vulnerabilities arising due to missing sequence checks. It can identify flaws in sequence checks made through session tokens and not through any other mechanisms such as CSRF tokens. TamperProof (Skrupsky et al. 2013) blocks workflow violation attacks arising due to failure in validation of session tokens used for maintaining the sequence of operation. Figure 2.6 provides a taxonomy for research articles addressing workflow vulnerabilities.

2.2.3.4 Business Logic Preservation in eCommerce Applications

The recent research community has started focusing on identification of logic vulnerabilities in eCommerce 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 (Pellegrino and Balzarotti 2014; Sun et al. 2014; Wang et al. 2011).

Wang et al. (2011) 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 (Wang et al. 2012) uncovers the critical logic flaws in applications integrating their services with the APIs of Google, Facebook, PayPal, etc. While the above works (Wang et al. 2012, 2011) analyze security of the applications, InteGuard (Xing et al. 2013) 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 and Balzarotti (2014) 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. (2014) 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 the currency.

Table 2.4 presents a summary of the research works addressing logic vulnerabilities, specifies whether the prototype focuses on detection/prevention of attacks, and highlights the type of coding flaws (discussed in detail in Section 4.1) detected by the prototype.

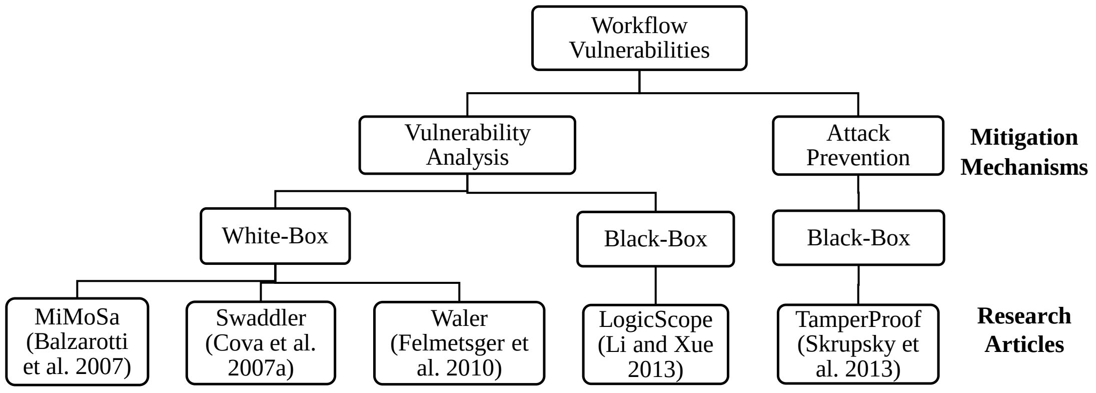
Workflow Vulnerabilities
Vulnerability Analysis
Attack Prevention
Mitigation Mechanisms
White-Box
Black-Box
Black-Box
MiMoSa (Balzarotti et al. 2007)
Swaddler (Cova et al. 2007a)
Waler (Felmetsger et al. 2010)
LogicScope (Li and Xue 2013)
TamperProof (Skrupsky et al. 2013)
Research Articles

Figure 2.6: Research on workflow vulnerabilities

Table 2.4: Summary of the literature on logic vulnerabilities

Research Article	System Type	Analysis	Type of Flaws							Remarks
			MAC	IAC	PO	MSV	SeV	StV	SO	
Swaddler (Cova et al. 2007a)	AP	B						✓		Involves code instrumentation
Nemesis (Dalton et al. 2009)	AP	B	✓	✓						Imparts performance overhead
BLOCK (Li and Xue 2011)	AP	B					✓	✓		Offline-learning and runtime protection, Imparts performance overhead
SENTINEL (Li et al. 2012)	AP	B						✓		Offline-learning and uses proxy to capture queries, Blocks malicious queries
Bypass-Shield (Mouelhi et al. 2011)	AP	B				✓				Detects MSV related to SQL injection and XSS
TamperProof (Skrupsky et al. 2013)	AP	B				✓	✓			Involves code instrumentation, Solution deployed in proxy
Flowwatcher (Muthukumaran et al. 2015)	AP	B	✓	✓						Proxy-based solution
MiMoSA (Balzarotti et al. 2007)	VD	W						✓		
Waler (Felmetsger et al. 2010)	VD	W						✓		Uses Symbolic model checking to identify logic flaws
Sun et al. (2011)	VD	W	✓							Requires developer to specify set of roles and their entry points
RoleCast (Son et al. 2011)	VD	W	✓							
SaferPHP (Son and Shmatikov 2011)	VD	W	✓							
WAPTEC (Bisht et al. 2011)	VD	W				✓				Employs symbolic evaluation

Continued on next page

Table 2.4 – Summary of the literature on logic vulnerabilities (continued from previous page)

Research Article	System Type	Analysis	Type of Flaws							Remarks
			MAC	IAC	PO	MSV	SeV	StV	SO	
ViewPoints (Alkhalaf et al. 2012)	VD	W				✓				Detects MSV related to SQL injection and XSS
FixMeUp (Son et al. 2013)	VD	W	✓							Repairs the access-control flaws
MACE (Monshizadeh et al. 2014)	VD	W	✓	✓						Requires annotations from the developers
ASIDE (Zhu et al. 2015)	VD	W	✓	✓						Employs interactive static analysis
LogicPatcher (Monshizadeh et al. 2016)	VD	W	✓	✓		✓				Repairs the logic flaws
NoTamper (Bisht et al. 2010a)	VD	B				✓				Given a web page, identifies client-server inconsistencies
PAPAS (Balduzzi et al. 2011)	VD	B			✓					Identifies parameter overriding vulnerabilities
LogicScope (Li and Xue 2013)	VD	B	✓	✓			✓			Identifies sequence violation due to missing validation of session variables
BATMAN (Li et al. 2014)	VD	B	✓	✓						Uses HTTP proxy to intercept HTTP packets and SQL proxy to intercept SQL queries
Scout (Wen et al. 2016)	VD	B	✓	✓						Addresses MongoDB applications

AP – Attack Prevention, VD – Vulnerability Detection, W – White-Box, B – Black-Box,
MAC – Missing Access Check, IAC – Incomplete/Improper Access Check, PO – Parameter Overriding,
MSV – Missing Server-Side Validation, SeV – Sequence Violation, StV – State Violation, SO – Session Variable Overloading

2.2.3.5 Research Challenges and Directions

In the recent decade, the focus of the security practitioners is shifting towards identification of business logic vulnerabilities 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 SQL injection and XSS attacks. Most of the existing works address only a specific type of logic vulnerabilities, such as parameter tampering (Alkhalaf et al. 2012; Balduzzi et al. 2011; Bisht et al. 2010a, 2011; Mouelhi et al. 2011; Skrupsky et al. 2013), and access-control vulnerabilities (Balzarotti et al. 2007; Cova et al. 2007a; Dalton et al. 2009; Felmetsger et al. 2010; Li et al. 2014; Li and Xue 2011; Li et al. 2012; Monshizadeh et al. 2014; Parno et al. 2009; Son et al. 2011, 2013; Sun et al. 2011). Therefore, development of approaches that are capable of identifying all the three types of logic vulnerabilities is an open area of research. Existing works for ensuring business logic in web applications take into consideration either the data flow or control flow of the application to identify vulnerabilities and attacks. Therefore, it would be promising to consider both the flows to increase the precision of the system for identifying vulnerabilities/attacks.

The absence of intended business logic specification is the major trouble in tackling logic vulnerabilities. The major contributing reason for the inability of existing vulnerability scanners towards detecting logic vulnerabilities is the absence of a general and automated approach for characterizing the application logic. The recent approaches proposed for detecting logic vulnerabilities try to extract the intended behavior of the application in any one of the following ways: (i) allow a tester to navigate through the web application for extracting the control-flow of the application (Li and Xue 2011; Li et al. 2012; Pellegrino and Balzarotti 2014), (ii) analyze the client-side and server-side code to extract the restrictions on parameters and identify data flow within the application (Bisht et al. 2010a, 2011; Skrupsky et al. 2013), and (iii) crawl through the application for exploration of the web pages (Balduzzi et al. 2011; Li et al. 2014). 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. Therefore, it would be interesting to pursue research in inferring the business specifications of the web application using a fully-automated approach, and developing secure frameworks to avoid inconsistencies between client and server-side validations.

2.2.4 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 (Shura 2010). Acunetix Web Vulnerability Scanner (WVS), Burp Suite, HailStorm (currently called Trustwave App Scanner (Trustwave)), HP WebInspect, IBM AppScan, McAfee Secure, MileScan ParosPro, N-Stalker, NeXpose, NTOSpider (currently acquired by Rapid7 and known as AppSpider[5]), 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 assessment of the applications. Table 2.5 highlights the capability of scanners towards detection of SQL injection, XSS and Business Logic Vulnerabilities. The column "Type" in the Table 2.5 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)).

[5]https://www.rapid7.com/products/appspider/

Table 2.5: List of commercial and open-source scanners and their capabilities

Company	Scanner	Type	SQLI		XSS			Business Logic		
			FOI	SOI	R	S	D	PM	ACV	WF
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	✓	×	✓	✓	×	×	×	×

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

Table 2.5 shows that the scanners are capable of detecting first-order SQL injection, reflected and stored XSS vulnerabilities. However, none of the listed scanners are capable of detecting second-order SQL injection and business logic (workflow) 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 (Bau et al. 2010, 2012; Doupé et al. 2010; Fonseca et al. 2007). 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. Additionally, 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 they are not able to detect logic flaws, as inferring the business specifications for different types of applications in an automated fashion is extremely challenging.

2.3 RESEARCH GAPS

Logic vulnerabilities allow the attackers to compromise the intended behavior of the application, and hence logic vulnerability identification requires extraction of the intended behavior of the application. The intended behavior is utilized for generating malicious requests that assist in detecting the vulnerabilities. The existing approaches take into account the parameters flowing to a web page, session variables associated with a web page, and the session variables used for maintaining the sequence of operations for modeling the intended behavior of the application to identify parameter manipulation, access-control, and workflow vulnerabilities respectively. The black-box approaches for detecting parameter manipulation consider parameters that appear in a request for a web page and do not take into account the interaction between multiple web pages (i.e., data flow), and therefore fail to identify few vulnerabilities (Bisht et al. 2010a). Literature on workflow vulnerabilities identification considers session variables used for maintaining the sequence of operations, and does not take into account other parameters such as CSRF tokens (Li and Xue 2013). The sequence of operations within the application is referred as workflow or control flow. The frameworks that identify access-control vulnerabilities arising due to inappropriate definition and validation of session variables, are not flexible enough to identify other types of session related vulnerabilities leading to session puzzling and session fixation attacks. The works by Bisht et al. (2010a), Li and Xue (2013), and Li et al. (2014) are ad-hoc approaches, and hence are unable to cover a wider range of logic vulnerabilities. Therefore, there is a need for a system that is capable of detecting a wide range of logic vulnerabilities by taking into account both the data flow and control flow information.

In the case of injection vulnerabilities, a large body of literature exists for preventing SQL injection, whereas only a limited number of articles exist for identifying XML injection vulnerabilities. Existing approaches and vulnerability detection tools available for addressing XML injection concentrate on detection/prevention of vulnerabilities/attacks in web services, and cover certain types of XML injection attacks only (Antunes et al. 2009; Antunes and Vieira 2011; Asmawi et al. 2012; Mitropoulos et al. 2009). None of the existing approaches and tools detect XML vulnerabilities in web applications using NXDs, and the XML injection vulnerability detection systems utilize attack vectors which are framed manually after a sound analysis of the literature. Therefore, there is a need for an approach that is capable of generating different kinds of attack strings for the detection of different types of XQuery injection vulnerabilities in web applications driven by NXDs.

2.4 SUMMARY

The omnipresence of web applications makes it imperative to ensure that the applications are secure, correct and efficient. This chapter has provided a comprehensive review of recent advances in securing web applications from the injection, business logic and session management vulnerabilities, and points out the unresolved issues that need to be addressed. It captures the current state-of-the-art and contributes a body of knowledge to the field of securing web applications by (i) discussing the various kinds of vulnerabilities that require major attention, (ii) illustrating the approaches suggested for detecting and preventing vulnerabilities, and countermeasures for detecting and preventing attacks on the application, (iii) highlighting the research challenges and future research directions, and (iv) pointing out the capabilities and limitation of the vulnerability scanners available for evaluating web applications.

CHAPTER 3

PROBLEM DESCRIPTION

As web applications get increasingly complex to support sophisticated business functionalities, logic and XML injection vulnerabilities have attracted increasing attention in recent years. Exploitation of these vulnerabilities in web applications leads to hazardous actions such as economic and privacy losses to the business enterprise and consumer respectively. Therefore, it is essential to proactively identify these vulnerabilities before attackers can leverage them. Existing literature employs white-box and black-box approaches for vulnerability detection. While white-box approaches are technology dependent and analyze the source code of the application, black-box approaches are technology independent and penetrate the application with malicious inputs for the detection. However, the existing black-box approaches on vulnerability detection face a major challenge in effective generation of malicious inputs/requests based on the behavior of the application for simulating attacks. Therefore, this work aims at developing a behavior-based approach for generation of attack requests and malicious user-inputs through black-box analysis to identify logic and injection vulnerabilities in web applications.

The first objective of this work is to develop a behavior-based approach for generation of attack requests to identify logic vulnerabilities in web applications. Logic vulnerabilities allow the attackers to compromise the intended behavior of the application, and hence generation of attack requests for logic vulnerability identification requires modeling the intended behavior of the application. The intended behavior of the application can be modeled by taking into account the parameters flowing to a web page, session variables associated with a web page, and the session variables used for maintaining the sequence of operations. The flow of parameters and session variables mark the data flow, and the sequence of operations between the web pages mark the control flow of the application. Existing approaches take into consideration either the data flow or the control flow to model the behavior of the application, and do not address detection of all types of logic vulnerabilities. Therefore, by considering the pros and cons of the state-of-the-art approaches, this work constructs a web application model reflecting the intended behavior in terms of both the data flow and the control flow. The model can be subsequently used for generating attack requests/vectors to detect all the three types of logic vulnerabilities.

The second objective of this work is to develop an approach for generation of attack requests/vectors to identify XQuery injection vulnerabilities in web applications driven by NXDs. The existing literature and tools identify XML injection flaws in web services, and there are no mechanisms for detecting XML injection in NXDs. Additionally, the XML injection vulnerability detection systems utilize the attack vectors which are framed manually after a sound analysis of the literature. Therefore, there is a need for an automated approach that is capable of generating different kinds of attack vectors for the detection of XQuery injection vulnerabilities in web applications driven by NXDs. The approach should be independent of the technology used for implementing the application, the functionality of the application, and should offer flexibility to identify new kinds of attacks when discovered. Hence, this work focuses on development of a context-free grammar to generate different kinds of XQuery injection attack vectors for the detection of XQuery injection vulnerabilities in native XML database-driven web applications.

To summarize, the primary objective of this work is to propose a behavior-based approach for generating attack vectors through black-box analysis for the detection of logic and XQuery injection vulnerabilities in web applications. It is further subdivided as follows:

(i) Constructing a behavioral model that reflects the data flow and control flow of the web application for generation of attack vectors to identify business logic vulnerabilities in web applications, and evaluating the effectiveness of the constructed model.

(ii) Proposing an attack grammar for launching attack vectors to identify different types of XQuery injection vulnerabilities in native XML database-driven web applications, and evaluating the effectiveness of the proposed grammar.

CHAPTER 4

ATTACK GENERATION FOR DETECTING LOGIC VULNERABILITIES

Logic vulnerabilities are unique to each custom application, potentially very damaging, and difficult to identify as the identification requires knowledge on the application behavior. These vulnerabilities often arise due to the following coding flaws: server-side validations, access checks, and sequence checks that are either missing or improperly implemented within the application. These coding flaws are to be identified for detecting the three types of logic vulnerabilities: parameter manipulation, access-control, and workflow vulnerabilities. The user-input validation missing at the server-side could be identified after extraction of the validation available for the same input at the client-side, and by tracking the parameters flowing between the web pages of the application. The access and sequence checks incorrectly implemented could be identified by observing the flow of session variables and sequence of operations within the application. The parameters and session variables flowing between web pages mark the data flow, and the sequence of operations mark the control flow. Thus, the logic vulnerability identification requires both the data flow and control flow information for learning the behavior of the application. This work constructs a model reflecting the intended behavior of the application in terms of data flow and control flow. The data flow and control flow information are extracted from the execution traces of the web application. The model is later used for deriving constraints placed on parameters, session variables, and workflows, which are violated for generating attack vectors to identify the vulnerabilities.

This work is part of the Research & Development project supported by the Ministry of Electronics and Information Technology (MeiTY) (formerly known as Ministry of Communications and Information Technology, MCIT), Government of India, and the contributions towards this work are listed as follows:

(i) Construction of a model to reflect the intended functionality of the web application in terms of data flow and control flow,

(ii) Derivation of constraints from the model and generation of attack vectors that violate the derived constraints for identification of different types of logic vulnerabilities, and

(iii) Evaluation of the constructed model on vulnerable benchmark web applications.

The rest of the chapter is organized as follows. Section 4.1 describes the software defects paving way for logic vulnerabilities and attacks. Section 4.2 presents the problem description and illustrates the logic vulnerabilities existing in web applications with a motivating example. Section 4.3 presents the proposed solution. Section 4.4 highlights the setup used for testing the proposed solution, and discusses the experimental results. Section 4.5 summarizes the chapter.

4.1 LOGIC ATTACKS IN WEB APPLICATIONS

Logic vulnerabilities allow malicious users to inject data that induce a web application software to behave differently and exhibit unforeseen behavior against the intention of the programmer resulting in attacks. These vulnerabilities originate due to the following implementation flaws: (i) missing server-side validation, (ii) missing and incomplete access checks, (iii) overloading of session variables, and (iv) missing sequence checks. These flaws permit the malicious users to circumvent the expected functionality of an application, and hence prevention of logic attacks requires identifying logic vulnerabilities arising due to the aforementioned flaws which are discussed in detail below.

Missing server-side validation: A web application uses client-side scripting to process and validate the user-supplied input for quick processing and for bringing down the server-side loads. However, malicious users can circumvent the client-side validation either by disabling the JavaScript execution or by submitting malicious requests which tamper the parameters in such a way that the restrictions placed on the user-input at the client-side are violated. If the application has enforced the same set of restrictions at the server-side, then the tampered parameters would be rejected by the application. Otherwise, the parameters cause the application to behave in a fashion different from the requirement specification of the application. Thus, failing to enforce user-input validation at the server-side results in *parameter manipulation attacks* (Bisht et al. 2010a, 2011; Skrupsky et al. 2013).

Missing/Incomplete access check: Web applications use HTTP, a stateless protocol, that treats each web request and response in an independent fashion. Hence, to maintain the state, applications use sessions to denote the logged in status of a user, the privilege level (say, admin or normal user), the sequence of operations, and a number of other things. Sessions can be maintained in two ways - purely at the client-side by means of cookies, or a combination of server and client-side. To maintain sessions, it is expected that the application performs some checks on the session variables before permitting access to its privileged resources. For example, a basic access check in an application verifies whether the session variables are set or not for allowing users to access the home page of the application only after logging in. However, some pages may not perform these checks, and consequently a malicious user can gain access to these pages. Thus, missing access checks lead to *authentication/authorization bypass attacks* (Li and Xue 2011; Son et al. 2011, 2013; Sun et al. 2011).

Similarly, for a user to access the application, the session variables should be properly validated against the role of a user and the HTTP parameters. There may exist scenarios in which the application verifies whether the session variables or the HTTP parameters are set or not, but do not validate the values of the session variables against the role of the user and the HTTP parameters. The improper validation of session variables against the role and the HTTP parameters is termed as incomplete/improper access checks that lead to *vertical/horizontal privilege escalation attacks* (Li and Xue 2011; Monshizadeh et al. 2014).

Overloading session variables: Uncontrolled creation/population of session objects or usage of identical session variables at various application entry points is called overloading of session variables, and may lead to *session puzzling attacks* (Chen 2011). These attacks do not contain any malicious input. They are legal actions allowed by the web application, but when performed in a particular order compromise the intended functionality of the application. While exploiting session puzzles, the creation of session objects can be indirectly initiated, and later exploited by accessing a sequence of entry points (web pages, web services, remote procedure calls, etc.) in a certain order. Session puzzles enable adversaries to perform a variety of malicious actions such as bypassing authentication/authorization and elevation of privilege for users, and upset the normal execution of the application.

Missing sequence check: Web applications use sessions to maintain the sequence of operations within the application (Li and Xue 2011). In addition to session variables, CSRF tokens generated at sensitive/critical pages of the application for preventing CSRF attacks indirectly assist in maintaining the sequence of operations (Jovanovic et al. 2006a). CSRF tokens are stored either in session or as a cookie, and are validated against the HTTP parameters to prevent attacks (CSRF). A CSRF token is generated at the server-side of the application in a critical page, and is either stored in the session or issued to the client and set as a cookie. The CSRF token stored in the cookie/session is supposed to be validated against the CSRF parameter in the HTTP request of the web page that follows the critical page. Thus, CSRF token enables to maintain the sequence of operations within the application. If a web page that follows the critical page fails to validate the value of the CSRF token stored in the session/cookie against the HTTP parameter, then there is a possibility that the critical page can be bypassed by the attacker by directly placing request for the pages following the critical page. Thus, lack of validation of tokens issued for maintaining the sequence at the server-side results in *workflow bypass attacks*.

Table 4.1: Symbols and definitions

Symbol	Definition
W	Finite State Machine
Q	Set of states in W
δ	Set of transition functions in W
Σ	Set of input symbols associated with a transition $t \mid t \in \delta$
A	Set of annotation functions in W
P	Set of parameters passed along a request in W
S	Set of session variables associated with a transition $t \mid t \in \delta$
R	Set of roles inferred from the set S
q_0	Initial state in W
F	Set of final states in W
q_s	Source state in a transition $t \mid t \in \delta$
q_d	Destination state in a transition $t \mid t \in \delta$
G	Underlying Directed graph of W
G'	Underlying undirected graph of W

4.2 PROBLEM DESCRIPTION

Given a web application, the objective of this work is to develop a systematic approach that is capable of generating attack vectors for detecting business logic vulnerabilities prevailing in web applications, independent of the functionality of the application. Generation of attack vectors demands modeling the intended behavior of the application. The model should reflect the data flow and control flow information, which are inferred as constraints for generation of attack vectors. In the context of this thesis, the data flow refers to the parameters and session variables flowing between different web pages, and control flow refers to the sequence of operations between web pages of the application. The symbols used in this chapter are defined in Table 4.1.

Motivating Example

This section provides a running example to illustrate the various implementation flaws leading to the logic vulnerabilities in web applications. Listings 4.1 to 4.8 illustrate a sample web application intended for maintaining medical records of patients registered with the application. The application has six web pages: *Login, Forgot Password,* and *Index* pages which are common to all the users of the application, and *View, Create,* and *Delete* pages intended for users with specific roles. Three user roles exist in the application: *Patient, Physician,* and *Administrator*. The business requirements of the application specify the following rules: (i) a patient can view their own record, (ii) a physician can view records of their own patients, and can create new medical records, and (iii) administrator can view and delete record of any patient registered within the application, and can create new records.

The user is first presented with the *Login.php* page. After the user has provided valid credentials, the application redirects the user to *Index.php* page. The *Index.php* page is presented with three hyperlinks to *View.php, Create.php,* and *Delete.php* depending on the role of the logged-in user retrieved from the database. *View.php* is available for all the three roles, *Create.php* is meant for both *Physician* and *Adminstrator*, and *Delete.php* is intended for *Administrator* only. Listing 4.1 is a JavaScript file that validates the input supplied by the user at *Create.php* page.

```
function validateInput() {
  var age = document.getElementById("age").value;
  if(document.getElementById("fname").value=="") {
    alert("User name can not be blank");
    return false;
  } else if (age < 0 || age > 150) {
    alert("Age must be greater than zero and less than 150.");
    return false;
  }
  return true;
}
```

Listing 4.1: Validate.js

```
<?php
  if(isset($_POST["userid"]) && isset($_POST["password"])) {
    $login = validateUser($_POST["userid"],$_POST["password"]);
    if(!$login) {
      HTTPRedirect("Login.php");
    }
    $_SESSION["userid"] = $_POST["userid"];
    //Code to retrieve role of the user from the database
    .....
    $_SESSION["role"] = $role;
    header("Location:index.php?userid=".$_SESSION["userid"]);
  } else {
?>
    //HTML code to get the login credentails from the user
    .....
    <a href="ForgotPwd.php">Forgot Password</a>
<?
  }
?>
```

Listing 4.2: Login.php

```
<?php
  if(isset($_GET["userid"])) {
    $user = $_GET["userid"];
    print("<a href=View.php?userid=.$user>View Record</a>");
    if($_SESSION["userid"]==$user && ($_SESSION["role"]=="
  physician" || $_SESSION["role"]=="admin")) {
      print("<a href=Create.php?userid=.$user>Create Record</a>"
  );
    }
    if($_SESSION["role"]=="admin" ) {
      print("<a href=Delete.php?userid=.$user>Delete Record</a>"
  );
    }
  } else {
    die("Please login to the application");
  }
?>
```

Listing 4.3: Index.php

```
<?php
  if(isset($_GET["userid"])) {
    $user = $_GET["userid"];
    if(isset($_SESSION["userid"]) {
      if($_SESSION["role"]=="patient") {
        //Code to Retrieve and Display Record
        .....
      } else if($_SESSION["role"]=="physician" || $_SESSION["
   role"]=="admin") {
        if(isset($_POST["patientid"])) {
          $patientid = $_POST["patientid"];
          //Code to Retrive and Display Record
          .....
        } else {
?>
          //HTML Code to get the Patient ID
          .....
<?        }
      }
    }
  } else {
    die("Please login to the application");
  }
?>
```

Listing 4.4: View.php

```
<script type='text/javascript' src='Validate.js'>
</script>
<?php
  if(isset($_GET["userid"])) {
    $user = $_GET["userid"];
    if($_SESSION["userid"]==$user) {
      if($_POST["mode"]=="insert") {
        //Code to Insert Record
        .....
      } else {
?>
        <form action="Create.php?userid=.$user" onsubmit="
   return validateInput()" method="post">
        //HTML code to get inputs from user for creating record
        ......
          <input type="hidden" name="mode" value="insert">
        </form>
<?    }
    }
  } else {
    die("Please login to the application");
  }
?>
```

Listing 4.5: Create.php

```
<?php
  if(isset($_GET["userid"])) {
    $user = $_GET["userid"];
    if($_SESSION["userid"]==$user && $_SESSION["role"]=="admin"
   ) {
      if(isset($_POST["patientid"])) {
        HTTPRedirect("Confirm.php");
      } else {
        if(!isset($_COOKIE["csrf_token"])) {
          $token =  bin2hex(random_bytes(32));
          setcookie("csrf_token",$token);
        }
?>
        <form action="Delete.php?userid=.$user" method="post">
          //HTML Code to Get Patient ID
          .....
          <input type="hidden" name="token" value=<?php echo
   $token ?>>
        </form>
<?    }
    }
  } else {
    die("Please login to the application");
  }
?>
```

Listing 4.6: Delete.php

```
<?php
  if(isset($_GET["userid"])) {
    $user = $_GET["userid"];
    if($_POST["confirm"]=="yes") {
      if(isset($_POST["token"])) {
        //Code to Delete Record
        .....
      }
    } else {
?>
      //HTML Code to Get Confirmation from user
      .....
<?
    }
  } else {
    die("Please login to the application");
  }
?>
```

Listing 4.7: Confirm.php

```
<?php
  if(isset($_POST["userid"])) {
    $_SESSION["userid"] = $_POST["userid"];
    header("Location:PwdRecovery.php?userid=".$_SESSION["userid
  "]);
  } else {
?>
    <form action="ForgotPwd.php" id="Pwd" method="post">
      UserID: <input type="text" name="userid"><br>
      <input type="submit" value="Next">
    </form>
<? } ?>
```

Listing 4.8: ForgotPwd.php

The web application maintains two session variables $role$ and $userid$. The variable $role$ keeps track of the privilege of a user within the application, and the $userid$ variable keeps track of individual users by means of their user ID. The presented application inherits several vulnerabilities which are discussed in detail.

Bug 1: $Index.php$ page does not verify whether the session variable $userid$ is set or not. As a result, any malicious user can access *Index.php* page without logging into the application by providing a valid value for the parameter *userid* in the HTTP request, and hence, the user would be provided with hyperlink to view the medical record. Thus, the exploitation of this missing access check results in authentication bypass attack.

Bug 2: $View.php$ page checks if $\$\_SESSION[\text{“}userid\text{”}]$ is set or not, but do not validate the value of the parameter $userid$ in the request with the session variable. As a result, a user can view the medical record of another user by modifying the value of $userid$ in the request after logging into the application. This improper access check in the page would result in horizontal privilege escalation attack. The page should have checked if the user ID passed is the same as the session variable *userid* which is set.

Bug 3: $Create.php$ page checks whether the *userid* set in the session and passed in the request match, but fails to verify the role of the user having access to the web page. As a result, a user with lower privilege can create medical records by issuing a direct request to $Create.php$ page. This incomplete access check in the web page would result in vertical privilege escalation/authorization bypass attack. This could be eliminated by checking whether the value of the $\$\_SESSION[“role”]$ variable is equal to *admin* or *physician*.

Bug 4: $Confirm.php$ page gets the $userid$ from the request, but fails to verify the role of the user and the value of the session variables. As a consequence, any user with lower privilege can delete existing medical records by issuing a direct request to $Confirm.php$ page. This page misses the access check, and hence would result in vertical privilege escalation/authorization bypass attack.

Bug 5: The application defines a session variable $userid$ when a user logs in. Some pages in the application check whether the session variable $userid$ is set and is equal to the $userid$ parameter in the request. Now, there is a password recover facility in the web application. If a user has forgotten their password, the web application prompts them for their userid. Once the userid is entered, the application proceeds to display the subsequent pages, and stores the value of the userid in the session variable $userid$, which is the same as the session variable stored after logging in. Setting up the session variable in the $ForgotPwd.php$ page is a simple mistake, but can have unforeseen consequences. For instance, after entering the userid, the session variable is set. Hence, a malicious user can instantly browse all the pages in the web application which checks only for the session variable $userid$. This is an example for session variable overloading. This error can easily be avoided if session variable names are not duplicated.

Bug 6: $Login.php$ gets input from the user, validates the input, and redirects the user to $Index.php$. If the user-input supplied is not valid, then line 5 of Listing 4.2 redirects the end-user to the same page i.e., $Login.php$, but the server executes the rest of the code (from line 7) as it is not embedded within the else clause. In other words, even after supplying invalid credentials, the web page sets the session variables because of which all the web pages in the application could be accessed. This kind of vulnerability is called Execution After Redirection (EAR) vulnerability (Doupé et al. 2011; Payet et al. 2013). This error can be avoided by properly embedding the source code in appropriate blocks.

Bug 7: The web page $Create.php$ gets input from the physician or admin to create medical records. There exist a field Age in the form which confines its value between 0 and 150 using client-side scripting presented in Listing 4.1. A malicious user can bypass the client-side validation and submit a request for $Create.php$ page, with an invalid value, say, -5 for the parameter age. The parameter age is not validated at the server-side before insertion of the value into the database, and hence the intended behavior of the application is violated. Thus, missing server-side validation is exploited for launching a parameter manipulation attack.

Bug 8: The web page $Confirm.php$ validates whether the CSRF token is present in the HTTP request or not, but fails to verify the value of the CSRF token set in the cookie against the HTTP parameter $token$. Even though CSRF token is used for preventing CSRF attacks, it indirectly helps in maintaining the sequence of operations within the application. As a consequence, any user can delete existing medical records by issuing a direct request to $Confirm.php$ page by just appending the parameter $token$ in the request. Thus, this web page misses the CSRF token validation, and hence would result in a workflow bypass attack.

4.3 PROPOSED APPROACH

The proposed approach named *LogAttackGen* (Logic Attack Generator) identifies different types of logic vulnerabilities prevailing in web applications by formulating a model reflecting the data flow and control flow of the application for generation of attack vectors. The proposed approach involves the following steps: (i) Construction of a model reflecting the intended behavior of the application from the execution traces. (ii) Inference of intended behavior of the application in the form of constraints related to data (i.e., user-input parameters), access-control, and control flow of the application from the web application model. (iii) Generation of concrete attack vectors that violate the derived constraints for identifying the three types of logic vulnerabilities. The steps involved in identifying the logic vulnerabilities are elaborated in the following subsections.

4.3.1 Trace Collection

Traces are essential to construct the model for the web application under test. Manual traces are generated by allowing a tester to navigate through the application in a browser configured to use a proxy server. The proxy intercepts the HTTP requests/responses, and stores them for further analysis to construct the model and discover the vulnerabilities. In addition to collection of the web traffic requests, session variables which are essential for maintaining the state of the application are extracted. In order to achieve this, the proxy is configured with an extension module that is responsible for extraction of session information, and the same is discussed in Section 4.4.1. The parameters flowing between the requests and responses assist in the detection of parameter manipulation vulnerabilities, and session variables assist in the detection of access-control vulnerabilities. The sequence of operations within the web application aids in detecting the workflow vulnerabilities.

The traces can be generated either manually or in an automated fashion using a crawler. However, in this work user-generated traces are preferred to automated crawling due to the following reasons:

(i) There is a possibility of inferring false behavior of the application as intended behavior by the crawler. For instance, consider a web page in an application which is vulnerable to vertical privilege escalation attack. In the case of using an automated crawler for exploration of the application during learning, the crawler is able to visit the vulnerable page as a normal user itself, and hence would result in inference of a false condition stating the page can be visited by a normal user rather than a highly privileged user. As a result of this false learning, the vulnerability in the application would be missed from being identified. Thus, if the application is vulnerable, then the automated crawling would result in inference of false behavior resulting in false negatives.

(ii) There is a possibility of missing few vulnerable pages from being identified by the crawler as the crawler may not navigate through the application in a fashion intended by the programmer. For example, consider an eCommerce website which has the following web pages namely *Purchase, Add New Item, Edit Item, Remove Item*, followed by *Confirm Order,* and *Payment* pages. Assuming *Purchase* page has hyperlinks to *Add New Item, Edit Item,* and *Remove Item* pages, if the crawler visits *Edit Item* or *Remove Item* web page with no items in the cart, then there is a possibility that the subsequent pages *Confirm Order* and *Payment* web pages might not be processed completely. As these web pages are visited during *Edit Item* or *Remove Item* workflows, even when a new item is added through the *Add New Item* page, the workflow will not be completed as the crawler marks *Confirm Order* and *Payment* pages as already visited. This may result in false inference of behavior of the application.

(iii) For identifying workflow vulnerabilities, the major challenge involved is the extraction of all valid workflows in the web application. This is because, the web applications today change the state not merely due to click action on hyperlinks, but also due to events that are triggered on elements embedded within the web pages. It is difficult to crawl such applications as a crawler cannot initiate a JavaScript event. Therefore, this work makes use of traces generated manually.

The HTTP requests/responses extracted from the execution traces are fed as input for constructing the model to discover the vulnerabilities. In order to achieve better coverage, the traces should be generated for each role of the application.

4.3.2 Model Construction

As already described in Chapter 3, the intended behavior of the web application is to be modeled for identifying any deviation in the actual behavior to detect the different types of logic vulnerabilities. This section describes the conceptual model proposed for reflecting the intended behavior of the application, and the approach followed for constructing the model.

Conceptual Model: A Finite State Machine (FSM) is utilized for modeling the behavior of the application, as FSM is well suited for modeling the behavior of any system. The information needed for constructing the FSM are extracted from the execution traces. To identify the three types of logic vulnerabilities, the web application is modeled conceptually as an annotated FSM (W) with hextuple $(Q, \Sigma, A, \delta, q_0, F)$, where

- Q denotes a finite non-empty set of states, and the states are represented as DOM structure of the web pages with unique URL in the application.
- Σ represents a finite non-empty set of inputs. The input symbol which marks the transition to the next state in the FSM is represented as either the URL of the hyperlink clicked or the URL of the redirected web page of the application.
- A is a set of annotations, which are used for providing additional conditions that assist the transition from current state to the next state. An annotation is represented as a three tuple given by
  $[R, P, S]$, where
  - R represents the set of roles (i.e., privilege level of users) having access to the page,
  - P represents the set of HTTP parameters and their respective values that get passed along the request, and
  - S represents the set of session variables and their corresponding values
- δ is the transition function, which is defined as a mapping from $Q \times \Sigma \times A$ into Q. For example, $\delta(q_s, i, a) = q_d$ indicates that if the current state is q_s with input symbol $i(i \in \Sigma)$ under the annotation $a(a \in A)$, then there will be a transition from the current state q_s to the next state q_d. This FSM is referred to as an annotated FSM.
- q_0 marks the initial state, where $q_0 \in Q$. Initial state is the home page of the application from where the navigation starts.
- F represents the set of final states, where $F \subseteq Q$. Final states refer to the pages that mark the end of navigation within the application (i.e., usually the *logout* page).

The DOM structure of the web page with unique URL in the application is represented as a state in the annotated FSM. The edges are labeled using the URL, and the HTTP parameters and session variables that are input/set by the users and application. The URL on the labeled edge is treated as the input symbol, and the parameters and session variables are treated as annotation functions. The input symbols and the annotation functions at a particular web page decide the transition to the subsequent web pages. The parameters labeled as annotations on the edges of the FSM represent the data flow, the session variables and their values are used for inferring access-control policies, and the transitions from one state to another state represent the control flow of the application. The data flow and control flow are used for identifying parameter manipulation and workflow vulnerabilities respectively, and the session variables are used for identifying access-control vulnerabilities.

In an ideal scenario, if all the web pages in the application are browsed by the tester for all the roles and for all the possible workflows, then the motivating example discussed in Section 4.2 can be modeled as shown in Figure 4.1. The annotated FSM has

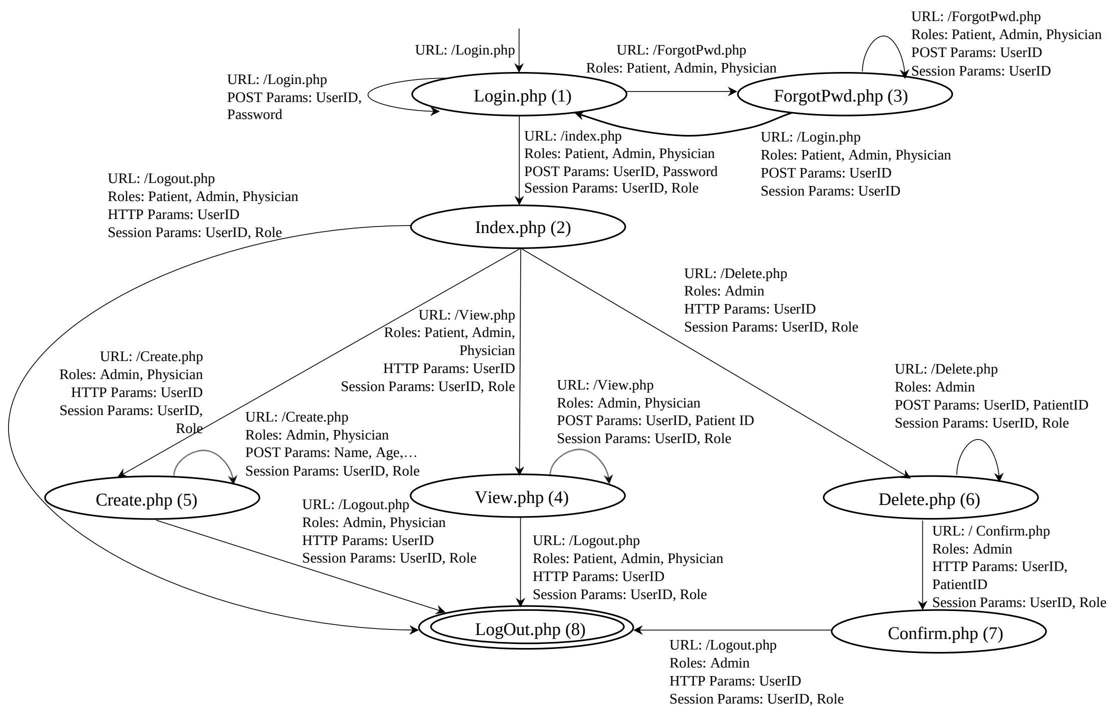

Figure 4.1: Web application model

eight states, and the input symbols represented on the edges in the state diagram mark the transitions within the application. $Login.php$ and $Logout.php$ mark the initial and final states of the application, respectively. The model reflects the intended business behavior of the application. *Login, Index, View, ForgotPwd,* and *Logout* pages are accessible by all users, while *Create*, and *Delete* pages are intended for admin, physician, and admin users, respectively. As seen in Figure 4.1, in web page $Index.php$, if the user clicks on the hyperlink for creating new medical records, then a transition to the next state, $Create.php$, occurs if the session variables $userid$ and $role$ are set, and the parameter $userid$ exists in the HTTP request. The input symbol $role$ marked on the edge represents the role for which $Create.php$ is available to the users.

Algorithm for Constructing the Proposed Model: Algorithm 4.1 presents the pseudocode for the construction of the annotated FSM from the execution traces. The algorithm analyzes each of the HTTP requests and the responses which are fed as input. Whenever a request is encountered, it is added to the input symbol set (Σ). The routine *ExtractParams()* retrieves the HTTP parameters from the request and stores them in the parameter set (P). On receiving a response, the algorithm verifies if the web page has been already visited. If not visited, then a new state is created and a transition is marked from the source state with the session variables and parameters associated with the transition added to the annotation set. The initial state and source state are initialized to null values before execution of the algorithm. The first state created during execution of the algorithm is assigned to the variable q_0. If the web page is already visited, then the state corresponding to that web page is retrieved using the function *getExistingState()*, and a new transition is created between the source state and existing state. If a transition already exists between the two states with different session variables, then the new session values are appended to the corresponding transition function. The new state created or the current state (i.e., the web page in which the user is currently in) becomes the source state for the next transition.

The session variables are extracted using the routine *ExtractSession()* implemented as part of “Session Extractor” module of the R&D project discussed in Section 4.4.1. The extracted variables are added to the set of session variables (S) in the annota-

Algorithm 4.1: Model construction

```
Data: Web Requests and Responses
Result: Model of the Web Application
$Q, P, S, \Sigma, \delta$ = []; $q_0$ = null; $q_s$ = null; //$q_s$ is Source state
while Input data is not empty do
    if request then
        Add request to $\Sigma$;
        if request contains parameters then
            $P$ = ExtractParams(request);
        end
    else
        //Code to handle HTTP Response
        if Web Page has not been visited before then
            $q_d$ = CreateNewState(); //$q_d$ is Destination state
            if $q_0$ is null then
                $q_0 = q_d$; //Initial state of the FSM is initialized
            end
            $t$ = CreateNewTransition($q_s, q_d, S, P$, request);
            Add $q_0, q_d$ to $Q$;
            Add $t$ to $\delta$;
        else
            //Web page already visited, get the state from $Q$
            $q_d$=getExistingstate($Q$);
            $t$ = CreateNewTransition($q_s, q_d, S, P$, request);
            if $t$ exists in $\delta$ with different Session Values then
                $t[session]$.add($S$); //Add new Session values to existing record;
            end
        end
        $q_s = q_d$;
        $S$ = ExtractSession();
    end
end
```

tion set (A) of the FSM. The routine identifies the server-side technology before extracting the session. For PHP applications, session information is usually stored in $/opt/lamp/temp/$ folder on a Linux installation, which is serialized and stored in the session file as variable name-value pairs. This information is deserialized and extracted for each request-response pair.

Figures 4.2 and 4.3 show the annotated FSM generated for the test applications, Scarf and Wackopicko respectively. It is not the complete FSM, and it shows only a few pages explored by the tester. The obtained FSM is visualized using Graphviz,[1] a graph visualization software.

4.3.3 Intended Behavior Extraction

The generated web application model (Annotated FSM) is analyzed to extract the intended behavior of the application in the form of constraints. The constraints are extracted by observing the states of the FSM, transitions, and the input symbol and annotations (i.e., URL, parameters and session variables) that have caused the transitions in the FSM. Three types of constraints are derived as follows: (i) Parameter constraints: The parameters causing transition from one state to another state, and the validation checks placed on the user-input parameters at the client-side are stored as parameter constraints. The validation checks on the user-input parameters are provided as input from a client-side analyzer discussed in Section 4.4.1. (ii) Access-control constraints: The destination state, parameters and session variables and their corresponding values leading transition to that state are utilized for deriving access-control constraints. (iii) Workflow constraints: The web pages that should not be skipped while following a workflow are identified from the control flow of the application, and are stored as workflow constraints. The approach adopted for extracting the three types of constraints is explained as follows.

Parameter Constraints: For the detection of parameter manipulation vulnerabilities, it is essential to extract constraints placed on the user-supplied input at the client-side so as to verify whether the same constraints are implemented at the server-side. Each web request expects a certain number and type of parameters / user-inputs that get passed with it. Each parameter has a name and a value, and the value may have some specific constraints it must satisfy. The constraints placed on the value of the parameters at the client-side are stored as parameter-related constraints. For example, an online banking application restricts the value of the field *amount* to be transferred within a range of $500 - 5000$. To identify the restrictions imposed on the parameters at the client-side of the web application, an HTML/JavaScript analyzer is developed as part of the R&D project. The HTML/JavaScript analyzer analyzes the client-side code of the web page

[1] http://www.graphviz.org/

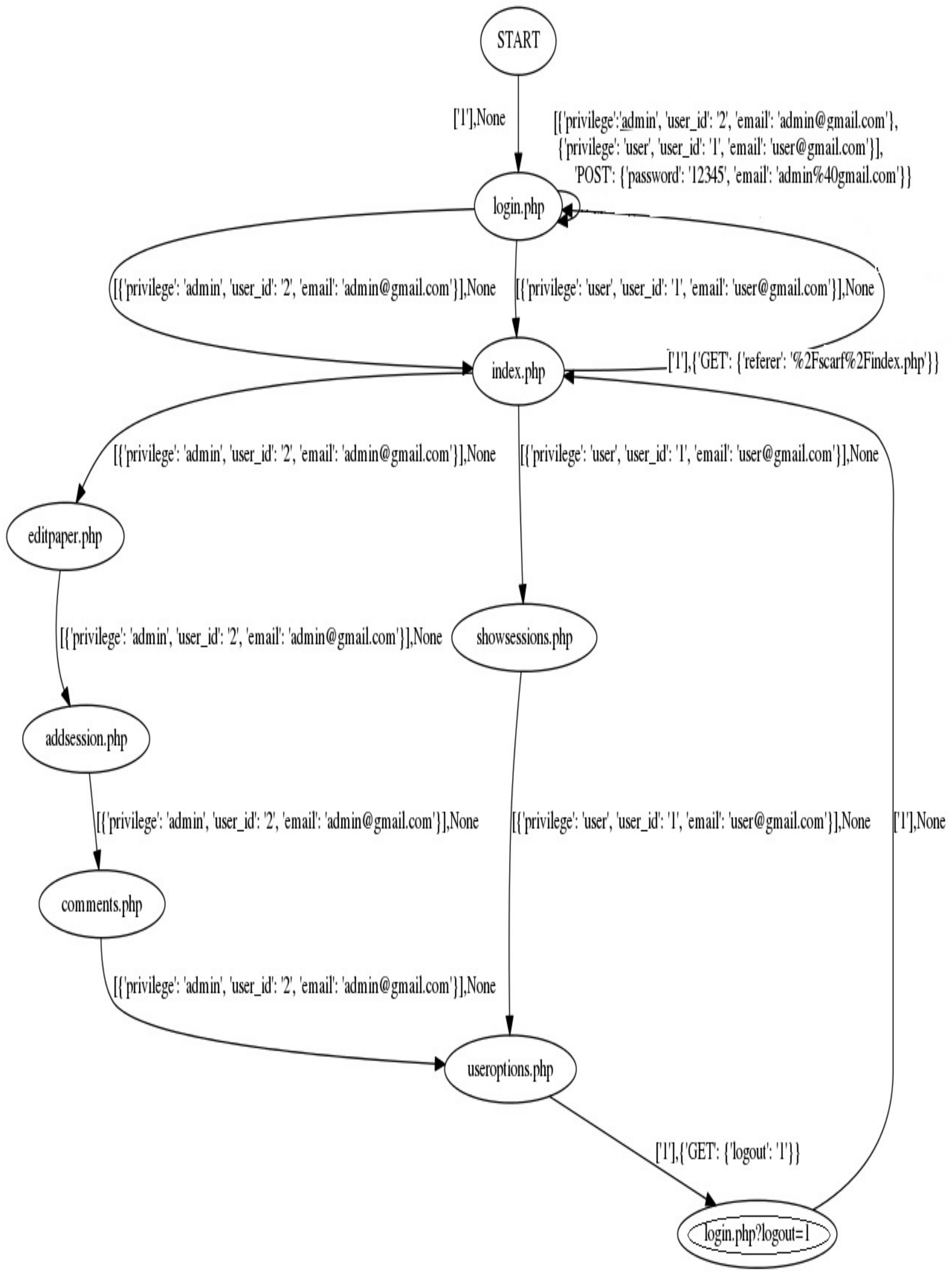
START
['1'],None
[{'privilege':'admin', 'user_id': '2', 'email': 'admin@gmail.com'}, {'privilege': 'user', 'user_id': '1', 'email': 'user@gmail.com'}], 'POST': {'password': '12345', 'email': 'admin%40gmail.com'}}
login.php
[{'privilege': 'admin', 'user_id': '2', 'email': 'admin@gmail.com'}],None
[{'privilege': 'user', 'user_id': '1', 'email': 'user@gmail.com'}],None
index.php
['1'],{'GET': {'referer': '%2Fscarf%2Findex.php'}}
[{'privilege': 'admin', 'user_id': '2', 'email': 'admin@gmail.com'}],None
[{'privilege': 'user', 'user_id': '1', 'email': 'user@gmail.com'}],None
editpaper.php
[{'privilege': 'admin', 'user_id': '2', 'email': 'admin@gmail.com'}],None
showsessions.php
addsession.php
[{'privilege': 'admin', 'user_id': '2', 'email': 'admin@gmail.com'}],None
[{'privilege': 'user', 'user_id': '1', 'email': 'user@gmail.com'}],None
['1'],None
comments.php
[{'privilege': 'admin', 'user_id': '2', 'email': 'admin@gmail.com'}],None
useroptions.php
['1'],{'GET': {'logout': '1'}}
login.php?logout=1

Figure 4.2: Annotated finite state machine for application Scarf

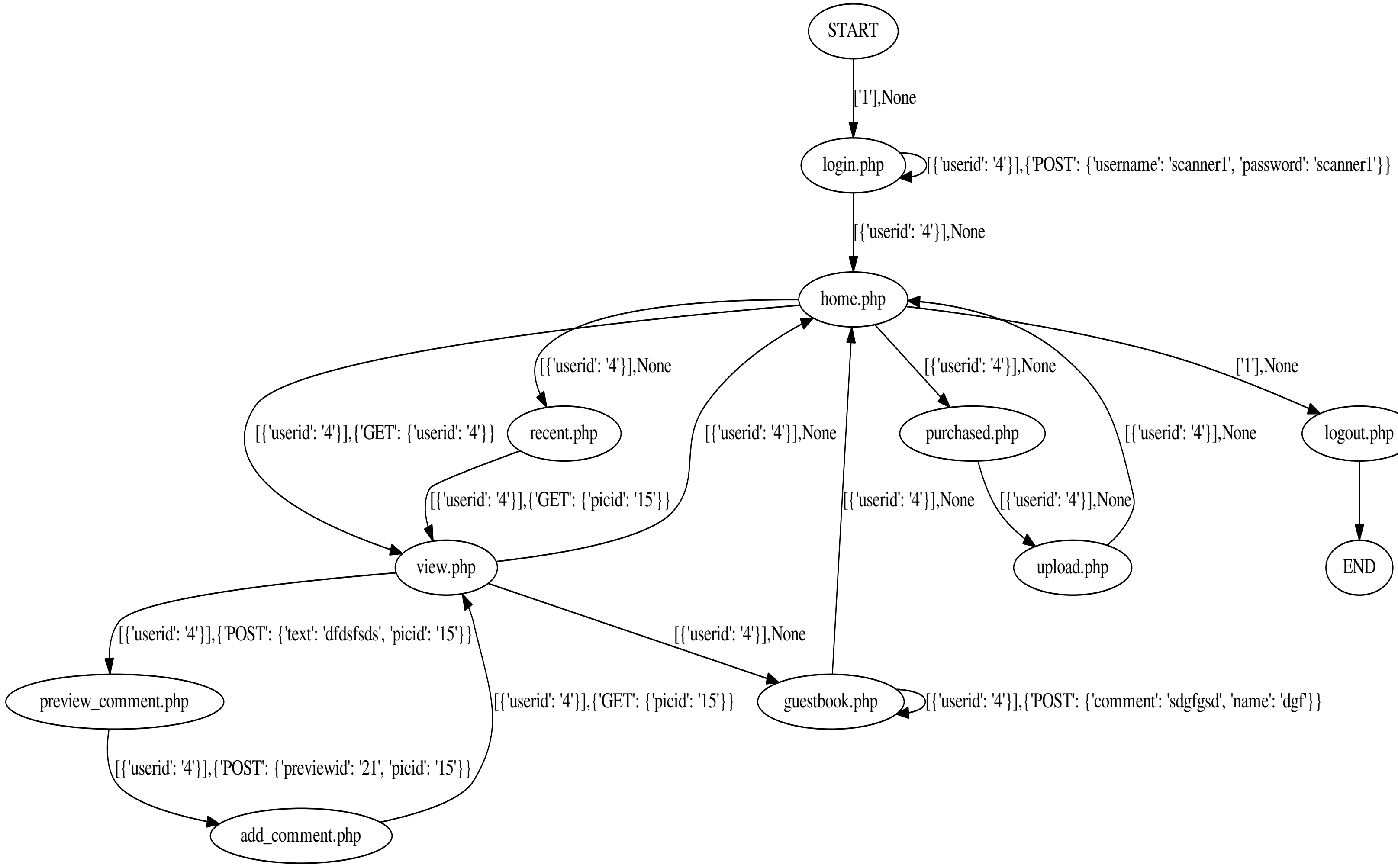
START
['1'],None
login.php
[{'userid': '4'}],{'POST': {'username': 'scanner1', 'password': 'scanner1'}}
[{'userid': '4'}],None
home.php
[{'userid': '4'}],None
[{'userid': '4'}],None
['1'],None
[{'userid': '4'}],{'GET': {'userid': '4'}}
recent.php
[{'userid': '4'}],None
purchased.php
[{'userid': '4'}],None
logout.php
[{'userid': '4'}],{'GET': {'picid': '15'}}
[{'userid': '4'}],None
[{'userid': '4'}],None
view.php
upload.php
END
[{'userid': '4'}],{'POST': {'text': 'dfdsfsds', 'picid': '15'}}
[{'userid': '4'}],None
preview_comment.php
[{'userid': '4'}],{'GET': {'picid': '15'}}
guestbook.php
[{'userid': '4'}],{'POST': {'comment': 'sdgfgsd', 'name': 'dgf'}}
[{'userid': '4'}],{'POST': {'previewid': '21', 'picid': '15'}}
add_comment.php

Figure 4.3: Annotated finite state machine for application Wackopicko

to extract constraints imposed (if any) on each field of the web form. The ‘constraints’ provide information about the possible values that a field in the form can hold, form fields which are mandatory, etc. The JavaScript analyzer provides three different types of constraints on the parameters flowing between web pages: (i) Data type constraint, (ii) Value constraint, and (iii) Length constraint. Data type constraint states the type of value that should be provided as input by the user. Value constraint refers to the value that an input field can hold. This constraint is represented in the form of a regular expression against the input field. Length constraint refers to the number of characters that an input can hold. These constraints are stored in a database for generation of attack vectors to identify input fields that are vulnerable to parameter manipulation attacks.

The constraints extracted using the JavaScript analyzer are mapped onto the web application model. For instance, for the parameter age discussed in Listing 4.5, the value constraint inferred is $0 < age < 150$. Similarly, for parameter $name$, the length constraint inferred is $Length(fname) > 0$. The inferred constraints are violated during attack generation for detection of parameter manipulation vulnerabilities. In case the response obtained for the attack request is similar to that of the normal request, then the web page is missing validation for the parameters at the server-side, and hence a vulnerability would be reported.

Access-control Constraints: For extraction of access-control constraints, the following types of web pages are to be identified by navigating through the FSM.

(i) Web pages that can only be accessed after logging in.

(ii) Web pages that are customized for individual users and should not be accessible to other users with the same privilege.

(iii) Web pages that can only be accessed by certain roles.

The aforementioned types of web pages can be identified from the observation of session variables and HTTP parameters and their respective values at each state of the constructed model. The flow of session variables and parameters across the states are inferred as constraints and are explained as follows. Firstly, the basic constraint inferred

at each state from the constructed model is the null check on session variables and parameters that have caused transition to that state. The second type of constraint inferred is an equality constraint derived from the session variable and HTTP parameter whose values are identical. The third constraint derived is based on the role of the user having access to the state. Some of the constraints derived from Figure 4.1 are as follows:
The constraints inferred at $Create.php$ (Listing 4.5) are:

(i) $\$\_SESSION[userid] \neq null$,

(ii) $\$\_POST[userid] \neq null$,

(iii) $\$\_SESSION[userid] == \$\_POST[userid]$, and

(iv) $\$\_SESSION[role] ==$ *Admin* || *Physician*

Similarly, $Delete.php$ (Listing 4.6) infers constraints such as

(i) $\$\_SESSION[userid] \neq null$,

(ii) $\$\_POST[userid] \neq null$,

(iii) $\$\_SESSION[userid] == \$\_POST[userid]$, and

(iv) $\$\_SESSION[role] == Admin$

The session variables and the parameters that check for null values (Constraints (i) & (ii)) help in identifying missing access check vulnerabilities. Constraint (iii) that checks the value of the session variable against the value of the HTTP parameters is useful in identifying horizontal privilege escalation vulnerabilities. The session variables that check their values against the roles (Constraint (iv)) registered with the application aid in the identification of authorization bypass vulnerabilities. Attack vectors are generated by violating each of the extracted constraints to identify vulnerabilities. To check if Constraint (i) is implemented, the page $Create.php$ is forcefully browsed with null value for the session variable $userid$. For Constraint (iii), the value of the parameter $userid$ in the HTTP request is modified to a value different from the session variable $userid$. In the case of Constraint (iv), requests for web pages $Create.php$, and $Delete.php$ are submitted with users having lower privileges (i.e., patient). In case the response obtained for the attack vectors being the same as that of normal requests, then vulnerability would be reported for these web pages. The details regarding attack generation are further explained in Section 4.3.4.

Workflow Constraints: With respect to detection of workflow vulnerabilities, the sensitive pages in the application are to be identified to verify whether they can be bypassed. In the context of this work, the sensitive/critical pages are defined as the web pages which are developed with the intention that they should not be skipped while following a workflow. The critical pages identified are stored as workflow constraints.

The critical web pages can be identified by modeling the web application as a directed graph. To achieve this, an underlying directed graph G is constructed from the annotated FSM. The states of the FSM (i.e., set of all web pages) represent the vertex set of G, and the edge set of G is obtained by joining two vertices (i.e., web pages), say u and v, by an arc if there exists a workflow from the vertex u to vertex v. Throughout this thesis, the directed graph so obtained is referred to as a control flow/workflow graph. The critical pages in the workflow graph are identified by identifying vertices whose removal disconnects the path to one or more vertices from the root node (i.e., the initial state of the FSM) of the directed graph. The identified vertices refer to web pages which disconnects the path to one or more subsequent web pages, when they are skipped from being visited. This concept may not identify web pages that are actually critical to the application in real-time. However, the identified non-critical pages are also tested for vulnerabilities, and hence the proposed approach avoids false negatives.

Additionally, this work identifies pages that are more sensitive to the application by identifying cut-vertices in the workflow graph. A cut vertex v of G is a vertex whose removal either increases the number of components of G or disconnects G. It also implies that the vertex v exists in all the paths from the vertex x to vertex y where $x \neq v$ and $y \neq v$. Thus, identifying cut-vertices helps in discovering web pages which disconnects the workflow, when skipped during navigation. It also implies that the web pages exist in multiple workflows, and hence are more critical to the application. For example, login page is a cut-vertex page, which exists in all the workflows in any service-oriented web application. The cut vertices can be identified by obtaining a simple undirected graph G' from the workflow graph G by ignoring the directions and self-loops. The directions are ignored because, the workflow graph is a weakly connected graph. Thus, a cut vertex of the undirected workflow graph corresponds to the web page developed with the intention that it should not be skipped while following the workflow. In other words, skipping that node or web page disconnects/disturbs the workflow of the application. Here, the set of cut vertices in the undirected workflow graph is obtained by performing a depth-first search on the graph.

The web pages corresponding to the aforementioned two conditions are stored as workflow constraints/critical pages. The workflows involving these critical web pages are considered for identifying workflow vulnerabilities. The request for the critical web page is skipped in a workflow to detect whether the critical page can be bypassed. Figure 4.4 represents the workflow graph obtained from the FSM shown in Figure 4.1. The critical pages identified from the graph are nodes (1), (2), and (6), which correspond to $Login(1), Index(2)$, and $Delete.php(6)$ pages in the application. Among these pages, $Login(1)$ and $Index(2)$ pages are the cut-vertex pages, and are more critical to the application.

As already described in Section 4.1, the sequence of operations within the application is either maintained using sessions or CSRF tokens. Identifying the critical pages in the application in the aforementioned fashion assists in unveiling pages that fail to

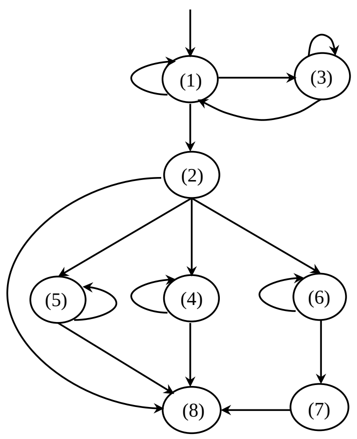

Figure 4.4: Workflow graph

Table 4.2: Bugs and corrections

Listing	Bug	Constraint	Location	Correction
Listing 4.3	Missing access check	$\$\_SESSION[userid] \neq null$	Line 4	Insert the statement $if(isset(\$\_SESSION[``userid"]))$
Listing 4.3	Incomplete access check	$\$\_SESSION[``userid"] == \$\_POST[``userid"]$	Line 8	Append the following condition $if(\$\_SESSION[``userid"] == \$\_POST[``userid"])$
Listing 4.4	Improper access check	$\$\_SESSION[``userid"] == \$\_POST[``userid"]$	Line 4	Modify the if condition to $if(\$\_SESSION[``userid"] == \$\_POST[``userid"])$
Listing 4.5	Incomplete access check	$\$\_SESSION[``role"] ==$ "physician" || "admin"	Line 6	Append the following condition $if(\$\_SESSION[``role"] == ``physician" \|\| \$\_SESSION[``role"] == ``admin")$
Listing 4.5	Missing server-side validation	$age > 0$ && $age < 150$	Line 9	Validate the input before inserting into the database
Listing 4.7	Missing access check	$\$\_SESSION[``role"] == ``admin"$, $\$\_SESSION[``userid"] ==$ $_POST["userid"]	Line 4	Insert the following statement $if(\$\_SESSION[``role"] == ``admin"$ && $\$\_SESSION[``userid"] == \$\_POST[``userid"])$
Listing 4.7	Missing sequence check	Delete.php (Critical Page)	Line 5	Append the following condition $if(\$\_POST[``token"] == \$\_COOKIE[``csrf\_token"])$
Listing 4.8	Session variable overloading		Line 3	Rename the session variable $userid$ to tmp_userid $\$\_SESSION[``tmp\_userid"]$

maintain the sequence due to lack of validation of both session variables and CSRF tokens. While lack of validation of session variables against the HTTP parameter could be identified using access-control constraints, pages that lack validation of CSRF token stored in the cookie could be identified only from the discovered critical pages in the application.

Thus, the annotated FSM is effectively used for extraction of parameter constraints, access-control constraints and workflow constraints, which are subsequently violated for discovering parameter manipulation, access-control and workflow vulnerabilities respectively. Table 4.2 describes the defects existing in the application discussed in Section 4.2, the inferred constraints and the corrections manually suggested to fix the flaws.

4.3.4 Attack Generation

The web application is penetrated with attack requests for identifying the vulnerabilities. The attack requests are submitted to the web application under test in such a way that the three types of constraints inferred from the FSM are violated. For example, if a web page requires session variables to be set, then attack request for the web page is submitted with null values for the session variables. Later, the response obtained is observed for reporting the different types of logic vulnerabilities. The algorithms used for generating attack vectors from the different types of constraints inferred are described in detail in this section.

Parameter Manipulation: The constraints extracted from the HTML/JavaScript analyzer are mapped onto the constructed FSM model and stored in a database. These constraints are later used for generating attack vectors to identify the vulnerabilities. The generation of attack vectors for identifying parameter manipulation vulnerabilities involves two cases: (i) providing values for parameters in such a way that the constraint imposed on the parameter is contradicted, and (ii) appending parameters which are not supposed to be present in a request for a web page. Algorithm 4.2 presents the pseudocode for generating the two types of attack vectors, and the details are described as follows:

- *Case 1: Contradicting the values of parameters:* The constraints against each form field stored in the database of the prototype serve as input to this algorithm. The constraints provide information regarding the type of constraint and the value, which are retrieved and violated to generate attack vectors. Each type of constraints imposed on the input field such as data type constraint, value constraint, and length constraint are violated for identifying vulnerabilities. For example, consider a form field *quantity* which is supposed to have an integer value within a range of $0 - 10$. In such cases, attack requests are submitted with the following malicious inputs: (i) a value less than zero ($quantity = -5$), and (ii) a value greater than ten ($quantity = 20$). Thus, each extracted constraint is violated for identifying the vulnerabilities.

Algorithm 4.2: Parameter manipulation attack generation

```
Data: Array of Input Constraints: constraint[]
foreach webpage in webpage_list do
    parameter_values = "";
    new_parameter = "";
    foreach form in the webpage do
        //Case 1: Contradicting the values of parameters;
        foreach input in the form do
            parameter_values.append(negateConstraint(constraint[input]));
            act_response = sendRequest(url, parameter_values);
            learnt_response = getValidResponse(url);
            if act_response = learnt_response then
                Report a vulnerability;
            end
        end
        //Case 2: Appending a new parameter;
        new_parameter.append(parameter);
        act_response = sendRequest(url, new_parameter);
        learnt_response = getValidResponse(url);
        if act_response = learnt_response then
            Report a vulnerability;
        end
    end
end
```

- *Case 2: Appending parameters to the HTTP request:* The constraints extracted from the JavaScript analyzer have information regarding the parameters that should flow from one page to another page, as well as the parameter that goes along a request only for a user with particular privilege. The constraint is violated by adding the parameter in the request of a user with lower privilege. From the motivating example discussed in Section 4.2, the parameter $mode$ with value as $insert$ in $Create.php$ is supposed to be present only for *physicians* and *administrator*. If the application is vulnerable, then the parameter $mode$ can be included in the HTTP request by a guest user of the application consequently permitting the guest user to create new records.

The response obtained during attack generation is stored and compared with the response obtained during the normal execution of the application. When a success

Algorithm 4.3: Authentication bypass attack generation

```
Data: $Q$ and $\delta$
Result: Vulnerability Report
forall t in $\delta$ do
    if t.Session != null then
        attack_response = SendRequest(t.destinationUrl);
        learnt_response = t.destinationUrl.dom;
        if attack_response = learnt_response then
            Report a vulnerability;
        end
    end
end
```

response is received for the attack vector, then the form fields submitted with contradicting values are reported as vulnerable injection points.

Access-control Violation: After extraction of access-control constraints from the model, the next step is to violate the constraints. The attack vectors generated for identifying access-control vulnerabilities are as follows:

Authentication Bypass: This attack focuses on identifying web pages that need to be accessed by privileged users, but fail to impose an access check in the source code. The constraints generated in the previous phase are used for identifying the web pages accessed by privileged users. Such web pages are accessed as a third-party/non-privileged user by submitting an HTTP request for the web page without submitting valid credentials. In other words, a web page in an application is accessed by any user without logging in. This is called authentication bypass attack. The responses obtained for the attack vectors are used for identifying the vulnerabilities. Algorithm 4.3 explains the steps followed for generating the authentication bypass attack. The set of states (Q) and transition functions (δ) are fed as inputs to the algorithm. The state set Q is utilized for identifying states that are accessed with session variables set in the transition function leading to that state. Requests are placed for those web pages without defining session variables, and the responses obtained are stored. The responses obtained for those requests are compared with the responses obtained during normal execution for reporting vulnerabilities.

Algorithm 4.4: Vertical privilege escalation attack generation

```
Data: $Q$ and $\delta$
Result: Vulnerability Report
1  //Step 1: Identify web pages and roles not having access to the page;
2  AttackSet=[];
3  forall t in $\delta$ with same t.DestinationUrl do
4      AttackRoles = SetofRoles - SetofAccessibleRoles;
5      AttackSet.add(t.DestinationUrl,AttackRoles);
6  end
7  //Step 2: Access web pages with non-accessible roles;
8  forall t in AttackSet do
9      forall role in AttackRoles do
10         Login(role);
11         attack_response = SendRequest(destinationUrl);
12         learnt_response = destinationUrl.dom;
13         if attack_response = learnt_response then
14             Report a vulnerability;
15         end
16     end
17 end
```

For example, considering the application Scarf shown in Figure 4.2, the web pages $addsession.php$ and $useroptions.php$ can be accessed by users after logging in. The proposed approach places requests for these web pages without setting proper user credentials to identify if the page is vulnerable to authentication bypass attack.

Vertical Privilege Escalation: This attack focuses on identifying the web pages with incomplete access checks. To identify such cases, highly privileged web pages are accessed by putting forward an HTTP request for the web page with a valid username in a parameter, but without proper session variables set for the privilege. This type of attack is called vertical privilege escalation attack. Algorithm 4.4 provides the pseudocode for generation of the attack, and it operates in two steps. The first step is to identify privileged pages and the roles not having access to the respective pages. The next step is to access the privileged pages with non-accessible roles. The algorithm is explained in detail as follows. The roles having access to a web page are extracted from the session variables, and stored in a list named *SetofAccessibleRoles*. Exclusion of *SetofAccessibleRoles* from the total set of roles intended for the application gives the set of non-accessible roles *(AttackRoles)* for the web page. The list *AttackSet* is updated with the URL of the web page and the set of roles not having the access to the page. The process is repeated until all privileged pages in the application are identified. At the end of the iteration, the list *AttackSet* consists of all privileged web pages in the application, and the set of roles not having access to the page. After identifying the privileged pages, attack requests are submitted for those privileged web pages with non-privileged roles. The responses obtained for those attack requests are compared with the responses obtained during normal execution for reporting vulnerabilities.

For example, considering the application Scarf shown in Figure 4.2, the web pages $editpaper.php$ and $comments.php$ can be accessed exclusively by an *admin* user. The attack generator module places HTTP requests for these web pages with the parameter $username$ set as a normal user to identify if the pages are vulnerable to vertical privilege escalation attack.

Horizontal Privilege Escalation: This attack focuses on identifying web pages with improper access checks. Such kind of vulnerabilities lead to horizontal privilege escalation attacks which mean a web page intended for a particular user can be accessed by any other user with the same privilege. In such cases, the web pages are accessed with the $username$ parameter set to any valid user registered with the application with the same role, and the responses are compared for reporting vulnerabilities. Algorithm 4.5 explains the steps followed for generation of the attack. The first step is to identify transition functions having identical values for any of the parameters and session variables. The second step is to place requests for the destination state of those transition functions with value set for the parameter different from the value of the session variable. The responses obtained for those attack requests are compared with the responses obtained during normal execution for reporting vulnerabilities.

For the application Wackopicko (Doupé et al. 2010), the web page $view.php$ intended for $user1$ is accessed by $user2$ by modifying the value of the parameter $userid$ to $user2$ in the HTTP request. If the response obtained for $user2$ is the same as the response received for the $user1$, then vulnerability will be reported.

Algorithm 4.5: Horizontal privilege escalation attack generation

```
Data: $Q$ and $\delta$
Result: Vulnerability Report
AttackSet=[];
forall t in $\delta$ do
    forall p in t.Params do
        forall s in t.Session do
            if p.value = s.value then
                AttackSet.add(t.DestinationUrl, p, s);
            end
        end
    end
end
forall t in AttackSet do
    Login(Session);
    p = ModifyParameter(Params);
    attack_response = SendRequest(destinationUrl,p);
    learnt_response = destinationUrl.dom;
    if learnt_response = attack_response then
        Report a vulnerability;
    end
end
```

Session Puzzling: In addition to the aforementioned cases, the proposed model identifies session puzzling (also known as session variable overloading) (Chen 2011), a kind of attack which targets the vulnerabilities in session variables. To the best of our knowledge, no prior work has been reported to capture the session variable overloading. To identify session puzzling, the proposed approach identifies transitions where the same session variables are set along different paths in the constructed FSM. This does not pose a problem, if the parameters passed to set the session variables are same in each case. A potential problem arises when the same session variables are set, but lesser information is applied to define it. Referring to Bug 5 discussed in Section 4.2, the session variable $username$ is set at $Login$ page using the user-input parameters $username$ and $password$. However, in $ForgotPwd$ page, the same session variable is set by accepting just the $username$. Again, this cannot be regarded as a vulnerability. This is because, other session variables may be set to maintain a state such that the user is not perceived as fully logged in. If requests are sent to pages where the $username$ param-

Algorithm 4.6: Session puzzling attack generation

```
Data: $Q$ and $\delta$
Result: Vulnerability Report
forall t in $\delta$ do
    prevtrans=getPreviousTransition(t);
    forall s in SessionVariables do
        if Session is set for t but not for prevtrans then
            param1=prevtrans.Params;
            SessionSetList=getSessionSetPoints(s);
            forall x in SessionSetList do
                param2=x.params;
                //Session set with lesser amount of information;
                if set(param1.keys)<set(param2.keys) then
                    Browse from start state to prevtrans;
                    Select dest which can only be visited when variable s is set;
                    attack_response = SendRequest(dest);
                    learnt_response = dest.dom;
                    if learnt_response = attack_response then
                        Report a vulnerability;
                    end
                end
            end
        end
    end
end
```

eter is necessarily set, and the response obtained is similar to the original responses, then this becomes a vulnerability. The aforementioned steps are explained through Algorithm 4.6, which presents the pseudocode for generating attack vectors to identify session puzzling vulnerabilities.

Workflow Bypass: The algorithm for initiating workflow bypass attacks requires the critical web pages in the application. As already described in Section 4.3.3, the critical web pages are pages that should not be skipped in a workflow. For each valid workflow with critical web pages, HTTP requests are placed sequentially for the pages in the workflow. However, the algorithm fails to place request for the critical web page and instead places request for the next web page (i.e., the page that follows the critical web page). Thus, skipping a web page from a valid workflow is called a workflow by-

Algorithm 4.7: Workflow bypass attack generation

```
Data: $Q$ and $\delta$
Result: Vulnerability Report
workflow_list = extractWorkflows($Q$, $\delta$);
foreach workflow in workflow_list do
    foreach node in workflow do
        if node not in criticalNodes_list then
            attack_response = sendRequest();
        end
    end
    learnt_response = getValidResponse(workflow);
    if learnt_response = attack_response then
        Report a vulnerability;
    end
end
```

pass attack. If the response obtained for the attack is the same as that of the response obtained during normal execution, then vulnerability is reported. The process is repeated for the other identified workflows and vulnerabilities are reported accordingly. Algorithm 4.7 presents the pseudocode for identifying workflow vulnerabilities. The list *criticalNodes_list* consists of the critical pages in the application.

For instance, the critical pages identified from Figure 4.4 are the nodes (1), (2), and (6). The workflows containing these pages in the application are: $(1) \longrightarrow (2) \longrightarrow (4) \longrightarrow (8)$, $(1) \longrightarrow (2) \longrightarrow (5) \longrightarrow (8)$, $(1) \longrightarrow (2) \longrightarrow (6) \longrightarrow (7) \longrightarrow (8)$, $(1) \longrightarrow (2) \longrightarrow (8)$, and $(1) \longrightarrow (3) \longrightarrow (1)$. Considering the path $(1) \longrightarrow (2) \longrightarrow (6) \longrightarrow (7) \longrightarrow (8)$, node (1) is skipped and request for page (2) is placed. If the request is successful, then node (2) is vulnerable to workflow bypass attack. The process is repeated for rest of the workflows to identify vulnerable web pages.

In the applications Scarf and Wackopicko, the critical web pages identified are $Login$ and $Index$, and $Login, Guestbook, View$ and $Preview\_comment$ pages respectively. The workflows (i.e., paths) containing these critical web pages are only considered for launching attacks.

4.4 EXPERIMENTAL STUDY AND ANALYSIS

This section provides details regarding the testbed on top of which the proposed model is implemented, and the applications used for evaluation. Extensive experiments are carried out to substantiate the effectiveness of the proposed approach which are discussed in detail.

4.4.1 Experimental Setup

The proposed approach *LogAttackGen*, for generation of attack vectors to identify logic vulnerabilities, is embedded within the prototype called DetLogic developed as part of the R&D project supported by MeitY, Government of India. DetLogic operates in three phases: (i) learning phase, (ii) attack generation phase, and (iii) discovery phase. Learning phase is meant for extraction of the intended behavior of the application. It extracts constraints imposed on parameters and constructs a model reflecting the intended workflow of the application. Attack generation phase is meant for generating concrete attack vectors that help in identifying the vulnerabilities in the application. In the discovery phase, the prototype attempts to detect vulnerabilities in the application by comparing the responses obtained during learning and attack generation phases. Figure 4.5 shows the high-level system design of DetLogic, and the components of *LogAttackGen* embedded within DetLogic. *LogAttackGen* forms the model generator and attack generation modules of DetLogic. The working principle of DetLogic is elaborated below.

Input: Seed URL and valid user credentials are given as input to identify the points of injection for parameter manipulation. Manual traces are generated by the tester for identifying access-control and workflow vulnerabilities.

Output: The output is a vulnerability report that lists the logic vulnerabilities, the types of attacks which are possible by exploiting the vulnerabilities, and the location of the vulnerabilities existing in the web application under test.

Trace Collection: Traces are the HTTP requests generated during navigation of the application under attack-free sessions. These traces can be generated either manually or automatically. The traces are analyzed for extraction of the intended behavior of the application. DetLogic makes use of traces generated both manually and automatically.

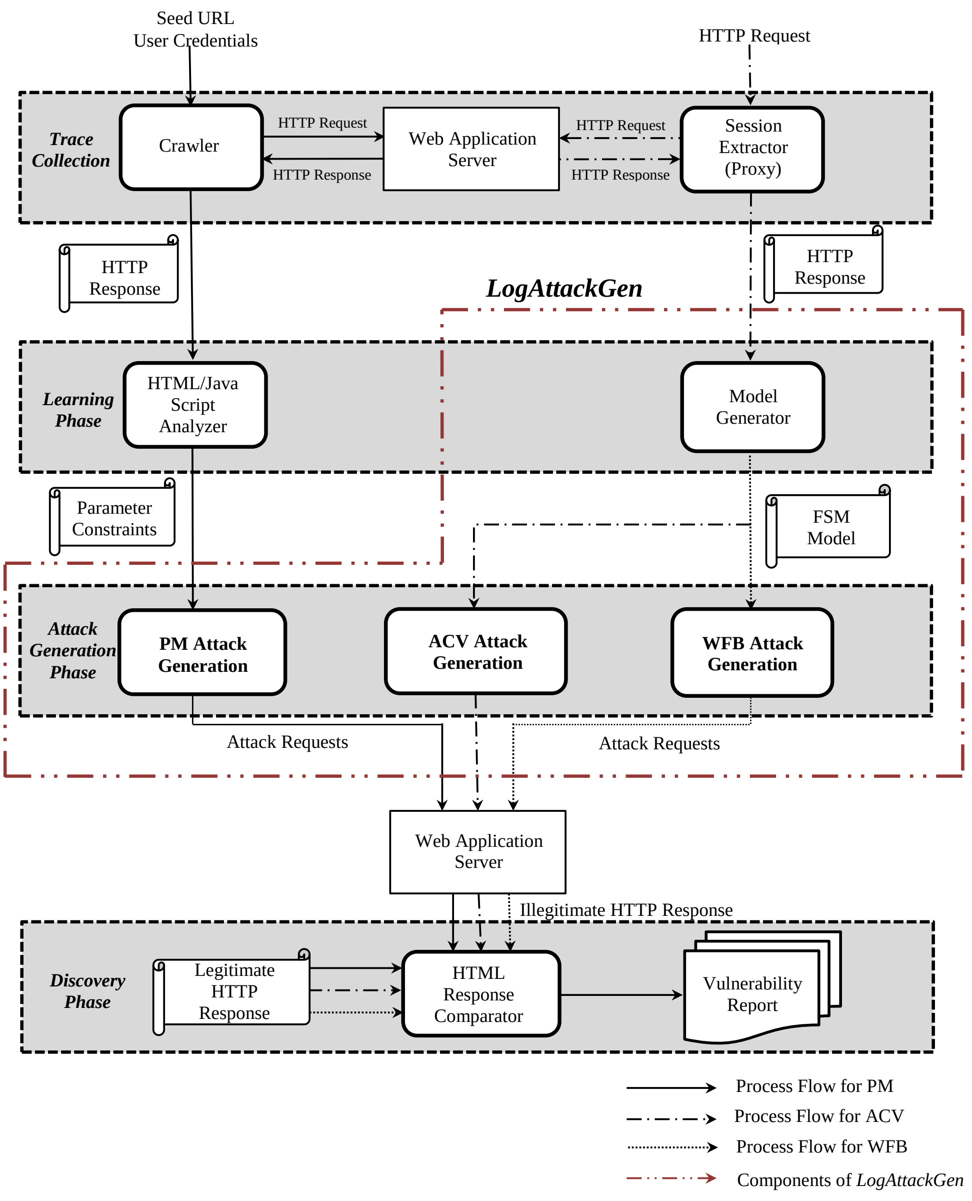

Figure 4.5: DetLogic architecture[2]

Manual traces are used for inferring the control flow of the application. The HTTP requests and responses generated during manual trace collection are intercepted and captured using a proxy. In addition to collection of the web traffic requests, session variables which are essential for maintaining the state of the application are extracted. These variables assist in the detection of access-control vulnerabilities. In order to achieve this, the proxy is configured with an extension module that is responsible for extraction of session information. Automated traces are used for detecting parameter

[2]The prototype is developed as part of R&D project supported by MeitY, Government of India.

manipulation vulnerabilities, and are generated using a crawler that explores all the web pages in the application on a depth-first search basis.

Learning: The learning phase is responsible for inferring the intended behavior of the web application. An HTML/JavaScript analyzer is employed for deriving the constraints on the parameters, and a model generator is utilized for extracting the intended behavior of the application in the form of an annotated FSM. The HTTP responses generated from the HTTP requests submitted to the application server during trace collection are collected, stored, and fed to the JavaScript analyzer and model generator. The analyzer and the model generator process the requests and responses, and represent the restrictions on parameters and navigation within the application in the form of parameter constraints and an annotated FSM respectively. The annotated FSM provides data flow and the control flow of the application.

Attack Generation: Three different types of attacks parameter manipulation, access-control violation and workflow bypass attacks are generated for identifying business logic vulnerabilities. Attack vectors are generated on injection points by either modifying values of parameters in the HTTP request in such a way that parameter constraints are violated, or appending parameters which are not supposed to be present in the request. For launching access-control violation attack, access-control constraints are used to identify whether highly privileged pages can be accessed by users with low privileges. To generate workflow bypass attack, the critical web pages in the web application are identified from the control flow, and are skipped from being visited.

Discovery: HTML response comparator is developed for comparing the response generated after each attack request against the corresponding response stored in the database during learning. Vulnerability is reported when the response generated during an attack is similar to that of the response obtained for a valid request. A report is finally generated with details of vulnerabilities, the types of attacks that exploited the vulnerabilities, and the flaws which led to the attacks.

Table 4.3: Applications for evaluation

Application	**Description**	**Vulnerabilities existing in the application**	**References**
Customized applications			
BookStore	Shopping application	Parameter manipulation vulnerabilities (Instrumented)	Halfond and Orso (2005)
Classifieds	Online management system for classifieds	Parameter manipulation vulnerabilities (Instrumented)	Halfond and Orso (2005)
Events	Event tracking system	Parameter manipulation vulnerabilities (Instrumented)	Halfond and Orso (2005)
Employee Directory	Online employee directory	Parameter manipulation vulnerabilities (Instrumented)	Halfond and Orso (2005)
Benchmark applications			
Scarf	Conference management system	Authentication bypass, Privilege escalation Vulnerabilities	CVE-2006-5909, Li et al. (2014); Li and Xue (2011, 2013); Li et al. (2012); Monshizadeh et al. (2014)
Wackopicko	Image management system	Access-control vulnerabilities	Li et al. (2014); Li and Xue (2011, 2013); Li et al. (2012)
OpenIT	IT management	Access-control vulnerabilities	Bisht et al. (2010a)
Puzzlemall		Session variable overloading	Chen (2011)
OsCommerce	eCommerce application	Workflow vulnerabilities (Instrumented)	Li and Xue (2011)

4.4.2 Test Applications

Table 4.3 presents the applications used for testing business logic vulnerabilities. The applications of Halfond and Orso (2005) such as BookStore, Classifieds, Employee Directory, and Events are instrumented with parameter manipulation vulnerabilities for the purpose of evaluation. The benchmark applications Scarf, Wackopicko (Doupé et al. 2010), OpenIT, and Puzzlemall (Chen 2011) are utilized for testing access-control vulnerabilities. The applications Scarf, Wackopicko, OpenIT, and OsCommerce are utilized for testing workflow vulnerabilities. The application OsCommerce is instrumented with a coding flaw in the web page *"checkout_payment"*, which fails to validate the value of CSRF token set in the cookie against the HTTP parameter. The proposed approach is implemented in Python, and Redis (a data structure server) is used to store the information regarding the web application under test for further processing.

4.4.3 Experimental Results and Discussions

The effectiveness of the proposed approach is demonstrated by testing it on the benchmark web applications. This section presents a discussion on the results obtained.

Parameter Manipulation Vulnerabilities: Table 4.4 shows the results obtained while testing the application for parameter manipulation vulnerabilities. The proposed approach is tested with four open-source web applications available in the test suite of Halfond and Orso (2005). These applications are instrumented with parameter manipulation vulnerabilities for the purpose of testing. The table gives the total number of forms existing in the application, the number of forms vulnerable to parameter manipulation vulnerabilities, number of vulnerabilities existing in the application and detected using the proposed approach. It can be inferred from the table that the proposed approach did not report any false positives. Table 4.5 gives the different types of constraints extracted for the test applications.

Access-control Vulnerabilities: Table 4.6 gives the results obtained while testing the application for access-control vulnerabilities. The proposed approach is tested with four open-source web applications. Three of these web applications have been used extensively in the literature as stated in Table 4.3. The fourth one is a relatively new open-source web application that exhibits session puzzling. As seen from the results, the approach works relatively well on most of the web applications. Table 4.7 shows the different types of access-control vulnerabilities reported using the proposed approach.

Table 4.4: Parameter manipulation vulnerabilities detected

Application	#URLs	#Forms	#Vulnerable Forms		#Vulnerabilities		#FP	#FN
			Existing	Detected	Existing	Detected		
BookStore	477	32	5	5	64	64	0	0
Classifieds	3,465	20	16	16	86	86	0	0
Events	67	12	11	11	39	39	0	0
Employee Directory	141	9	8	8	42	42	0	0

FP – False Positives, FN – False Negatives

Table 4.5: Number of constraints extracted

Application	#Constraints			
	#Length	#Data	#Value	#Total
BookStore	16	48	49	113
Classifieds	12	18	14	44
Events	0	12	8	20
Employee Directory	0	8	16	24

Table 4.6: Access-control vulnerabilities detected

Application	#LOC	#States	#Attack Requests	# Vulnerabilities		#TP	#FP	#FN
				Existing	Detected			
Wackopicko	4,037	12	9	4	3	3	0	1
Scarf	1,913	12	27	8	9	8	1	0
OpenIT	26,035	14	2	2	2	2	0	0
Puzzlemall	1,979	19	1	1	1	1	0	0

LOC – Lines of Code, TP – True Positives, FP – False Positives, FN – False Negatives

Table 4.7: Types of access-control vulnerabilities detected

Application	#Missing Access Check	#Incomplete Access Check	#Improper Access Check	#Session Variable Overloading
Wackopicko	2	0	1	0
Scarf	3	4	2	0
OpenIT	0	0	2	0
Puzzlemall	0	0	0	1

Discussion: One of the drawbacks of this approach is that the model is constructed using only a single trace generated by the user. Even though this is sufficient for identifying some kinds of attacks, this may result in false positives while detecting horizontal privilege escalation. This is due to the fact that the approach assumes the constraints based on the available traces. If only one trace is presented, then it can erroneously assume some constraints that are actually not constraints. This results in false positives in the case of the application Scarf. As an example, the $addsession.php$ page in Scarf application uses a parameter $month$ which is an integer value ranging from 1 to 12. Assuming the input provided by the tester for the parameter *month* is 5, and by chance the user logged in also has a session variable $userid$ set to 5, then the proposed approach erroneously infers a constraint stating that the value of the parameter $month$ and the session variable $userid$ should be same. Unfortunately, this is not true. This drawback can be removed by using multiple traces and stricter constraint extraction.

The proposed approach reports one false negative in Wackopicko. The reason behind this is that $highquality.php$ page in the application allows the attacker to view high-quality pictures without purchasing them by manipulating the $picid$ parameter in the request. This relationship between users and pictures is not incorporated as part of session variables and exists in the database table “own”. As a result, the proposed approach cannot capture the constraint when constructing the FSM model. Therefore, no attack vectors will be generated to violate such constraint, and hence is missed from being identified.

Table 4.8: Comparison of the proposed approach with LogicScope

	Proposed Approach						**LogicScope**					
Application	# States	# Attacks	# Vulnerabilities (E)	# Vulnerabilities (D)	# False Positives	# False Negatives	# States	# Attacks	# Vulnerabilities (E)	# Vulnerabilities (D)	# False Positives	# False Negatives
Wackopicko	12	9	4	3	0	1	2	21	4	2	0	1
Scarf	12	27	8	9	1	0	3	49	8	10	2	0
OpenIT	14	2	2	2	0	0	5	65	2	3	1	0
Puzzlemall	19	1	1	1	0	0	2	NA	1	0	0	0

E – Existing, D – Detected

Comparison with LogicScope: The results are comparable to existing work LogicScope (Li and Xue 2013). Table 4.8 gives the comparison of the results obtained for the proposed approach with the results as stated in LogicScope. Puzzlemall is not used in LogicScope, and hence the number of attack requests launched is marked as not applicable (NA). The number of states that would be inferred is two, since it has only two session variables. LogicScope is incapable of detecting the session puzzling vulnerability existing in Puzzlemall.

The model generated by LogicScope treats the cartesian product of the session variables and their values as states, while the proposed model considers the web pages as states of the FSM. In addition, the model proposed by LogicScope is not flexible enough to address vulnerabilities leading to session puzzling attack while the proposed model captures session puzzling. The proposed model is also adaptable to identify other types of vulnerabilities such as EAR vulnerability, and vulnerabilities arising due to improper handling of session identifiers.

Workflow Vulnerabilities: Table 4.9 gives the results obtained while testing the application for workflow vulnerabilities. The proposed approach is tested with four open-source web applications. Scarf and Wackopicko have $Login$ pages that can by bypassed by $Comments.php$ and $Generaloptions.php$, and $View.php$ and $Upload.php$, respectively. OsCommerce, a commercially used eCommerce application, is instrumented

Table 4.9: Workflow vulnerabilities detected

Application	#LOC	#States	# Vulnerabilities		#FP	#FN
			Existing	Detected		
Scarf	1,913	12	2	2	0	0
Wackopicko	4,037	12	2	2	0	0
OpenIT	26,035	14	0	0	0	0
OsCommerce	22,642	31	1	1	0	0

LOC – Lines of Code, FP – False Positives, FN – False Negatives

with a coding flaw in $checkout\_payment$ page, where the CSRF token set in the cookie at $checkout\_shipping.php$ is not validated against the HTTP parameter leading to a workflow bypass attack. As a result, a user could pay for the purchased items without inclusion of the shipping charges. The state machine inferred from the application with the role as customer has 31 states. Thus, the number of states inferred for a real time application is less than 50 due to consideration of the web page templates.

Discussion: While BLOCK (Li and Xue 2011) and LogicScope (Li and Xue 2013) detect workflow bypass resulting due to improper session management, the proposed approach is capable of detecting workflow bypass attacks resulting due to improper management of CSRF tokens as well. The correctness of results obtained using the approach depends on the model which is generated using manual traces. Therefore, to achieve better accuracy the tester need to explore all possible workflows.

Performance Evaluation: The performance of the proposed approach is evaluated using the following metrics: true positive rate (TPR), false positive rate (FPR), false negative rate (FNR), precision, and recall. The FPR and FNR of the proposed approach for both parameter manipulation and workflow vulnerabilities are 0%. With regard to detection of access-control vulnerabilities, the FPR and FNR are 5% and 6.25% respectively. The precision and recall percentage values are 99.1% and 97.9% respectively. The overall TPR of the proposed approach is found to be 97.9%.

4.4.4 Advantages and Limitations

The advantages of the proposed approach are as follows:

- The generated FSM represents every aspect of the interactions between the web application and the user. The annotations on the FSM offer the flexibility to identify the three different types of logic vulnerabilities.
- The session variables marked on the annotated FSM make the model capable and adaptable for identifying new types of vulnerabilities related to session management as compared to LogicScope (Li and Xue 2013). The adaptability offered by the model renders a way to generate attack vectors for detecting session puzzling vulnerabilities. Additionally, vulnerabilities such as improper handling of session identifiers (e.g., pages that issue session identifiers to users before logging in, but fail to modify them after logging in) can be identified by mapping the session identifier onto the model and triggering appropriate attack vectors.
- The FSM model is capable of generating attack vectors to identify workflow vulnerabilities arising due to CSRF token validation missing at the server-side which is not addressed in BLOCK (Li and Xue 2011) and LogicScope.
- The generated model can be used for detection of EAR vulnerabilities as well. The first type of access-control constraint inferred from the FSM (i.e., web pages at which session variables are defined and set) can be used for identifying EAR vulnerabilities. During attack generation, a request for the web page that defines the session variables is placed with invalid values for the input parameters. If the application is not vulnerable, then the request will be redirected to an error page. Consecutively, a second attack request is placed for the web page that follows the previous web page. If a success response is received, then EAR vulnerability is reported. Thus, the inferred constraint can be effectively used for identifying EAR vulnerabilities by placing two attack requests consecutively.
- The model offers flexibility in identifying the severity of vulnerabilities existing in the application. The cut-vertices identified from the control flow graph of FSM can be used to identify the severity of vulnerabilities. The out-degree of the cut-vertex node can be used for ranking the nodes which in turn would help in marking the severity of web pages in the application.

- Even though the number of states of the annotated FSM is more as compared to LogicScope, we tend to minimize the number of states by representing the states of the FSM using the web page template (i.e., DOM structure) after ignoring the data content in the web page. For any large application such as an eCommerce application, the size of the FSM will not increase with the number of web pages to display millions of products being sold. All these web pages will be treated as a single page while framing the state of the FSM.

Even though the proposed approach works based on a black-box approach in an automated fashion, few limitations exist which are listed as follows.

- The approach requires the session information at the server-side of the application for detection of access-control vulnerabilities.
- The correctness and completeness of the results depend on the model generated which in turn depends on the user navigating through the web application under test. When the user does not explore all the possible navigations within the application, the vulnerabilities in unvisited pages may be missed from being identified.

However, limitations regarding automated and manual trace collection can be overcome by using a simulated crawler. The simulated crawler requires a user to navigate through the application, and fill in all the forms in the application with valid values. The sequence of operations in the application and the possible values for form fields are recorded which can be later replayed during the learning phase.

4.5 SUMMARY

This chapter presents an approach for generation of attack vectors to detect three different types of logic vulnerabilities existing in web applications. The proposed approach employs black-box analysis for inferring the intended behavior of the application, and penetration testing and forceful browsing for assessing the application for vulnerabilities. The proposed approach extracts the data flow and control flow of the application by analyzing the execution traces. The data flow and control flow information are used for constructing an annotated FSM. The FSM is subsequently used for deriving parameter-related, access-related, and workflow constraints which are violated to generate attack vectors. The attack vectors generated by violating the parameter constraints tamper the parameters, and thus perform penetration testing to identify parameter manipulation vulnerabilities. The attack vectors generated by violating the access-control and workflow constraints forcefully browse the web pages to identify the potential access-control and workflow vulnerabilities in the application. In addition to the three types of vulnerabilities, the proposed approach is capable of detecting vulnerabilities leading to session puzzling attacks. The proposed approach has been exhaustively tested on benchmark PHP and Java web applications, and it is found to work effectively with a precision, and a true positive rate of 99.1% and 97.9%, respectively. Thus, it can become an essential tool for ensuring the security of applications against logic attacks irrespective of the technology used for implementing the application.

CHAPTER 5

ATTACK GENERATION FOR DETECTING XQUERY INJECTION VULNERABILITIES

XQuery injection vulnerabilities allow an adversary to inject malicious code as part of the user-input, which on execution results in extraction/insertion of unwanted information from/into the XML documents stored in NXDs. The major cause for these vulnerabilities is insufficient validation of user-input, and hence identification of these vulnerabilities demands probing the application with malicious inputs to ensure the availability of validation functions. If the malicious input is processed successfully by the application, then it implies the absence of validation functions, and hence a vulnerability is reported. Therefore, this work proposes an approach for crafting malicious inputs resulting in attacks for identifying different types of XQuery injection vulnerabilities in web applications driven by NXDs. The proposed approach formulates a context free grammar based on different types of attacks specified in OWASP (2015) guidelines for generating malicious inputs to detect the vulnerabilities. This work is part of the Research & Development project supported by MeitY, Government of India and the contributions towards this work are as follows:

(i) Development of an attack grammar to generate different types of XQuery attack strings so that all possible points of injection in the web application are captured by the proposed approach.

(ii) Identification of three new categories of attack vectors, alternate encoding, evaluation function, and XQuery comment injection attacks not listed in OWASP.

(iii) Evaluation of the proposed approach by testing on vulnerable, native XML database-driven web applications.

The rest of the chapter is organized as follows. Section 5.1 describes the importance of XML-based web applications and illustrates an XQuery injection attack with an example. Section 5.2 presents the problem description. Section 5.3 describes the proposed solution and discusses the new categories of XQuery injection attack vectors identified as part of this work. Section 5.4 presents the prototype on top of which the proposed solution is implemented, and discusses the experimental results. Section 5.5 summarizes the chapter.

5.1 XQUERY INJECTION IN NATIVE XML DATABASE

A lot of database-driven web applications in use today are XML-based as they involve exchange of information with multiple applications through XML documents. XML is a data representation that favors integration and interoperability between heterogeneous web applications. The information exchanged between the applications in the form of XML documents can be processed efficiently when they are stored appropriately. These documents are stored in either an extended relational DBMS or a native XML database system (Chaudhri et al. 2003; Liu and Murthy 2009). XQuery / XPath can be used as a query language for retrieving the data from XML documents.

A Native XML database (NXD) has XML document as its fundamental unit of storage, and defines a logical model based on the content in the XML document (Pavlovic-Lazetic 2007). NXDs are employed in cases where the data involved do not fit the relational data model, but fit the XML data model. NXDs are generally preferred for applications that hold highly diverse information, involve integration of information from different set of applications, and handle rapidly evolving schemas. NXDs are also preferred for applications that work with a huge set of documents or large-sized documents (e.g., books, web pages, marketing brochures), and involve management of long-running transactions like finance, pharmaceuticals, etc. (Bourret 2009). Use of relational databases and flat file systems for building such applications results in issues such as scalability and lack of structured queries. These issues can be overcome by using NXDs with XQuery/XPath as the query language for processing. Some of the popular NXDs are BaseX, eXistDB, and MarkLogic.

NXDs find applications in a wide variety of domains such as document management systems, healthcare systems, financial applications, business-to-business transaction records, catalog data, and corporate information portals (Staken 2001). Real-world business applications that employ NXDs to manage their content are Elsevier Science publishers, Las Vegas Sun publishers (Bourret 2009), the Tibetan Buddhist Resource Center (TBRC) (Siegel and Retter 2014), etc. The Tasmanian government websites use NXD for helping users to track legislation. NXDs are used to store various other types of documents such as drug information sheets, contracts, case law, and insurance claims. Commerzbank and Hewlett Packard use NXD for integration of information from a variety of sources to handle financial and business transaction data (Bourret 2009). Healthcare applications prefer to store electronic health records (EHR) in NXDs for efficient storage and retrieval of information from the available medical records (i.e., scan reports, prescriptions, etc.) (Al-Hamdani 2010; de la Torre et al. 2011; Lee et al. 2013).

The existing literature reveals that there is a growing demand towards usage of NXDs in web applications. Even though various XML security standards (Hirsch 2002; W3C 2015) such as XML Encryption, XML Digital Signature, and XML access-control markup language are defined for preserving confidentiality, integrity and access-control mechanisms of XML documents, when NXDs are used at the backend, any vulnerability in the source code of the application may allow an adversary to perform unwanted actions resulting in extraction/modification of information from/in the documents through the query language (i.e., XQuery/XPath). As the content of highly sensitive applications like finance, healthcare, etc. are driven by NXDs, security of NXDs is vitally important to ensure the integrity, privacy and confidentiality, and to make sure that information is used appropriately (Huang 2003). Hence, this work concentrates on identifying XQuery injection vulnerabilities in web applications driven by NXDs.

XQuery Injection

A simple example of an XQuery injection attack is described with the following XML document, *users.xml* (Auger 2010):

```
<?xml version="1.0"encoding="ISO-8859-1"?>
<userlist>
   <user category="group1">
      <fname>john</fname>
      <status>good</status>
   </user>
   <user category="admin">
      <fname>john</fname>
      <status>good</status>
   </user>
   <user category="group2">
      <fname>mary</fname>
      <status>good</status>
   </user>
</userlist>
```

The source code for retrieving the detail of a user from the XML document is given below.

```
String strName = (String) request.getParameter("name");
String strQuery = "xquery for $x in doc('users.xml')/userlist/user where $x/fname='"+
strName + "' return $x";
```

Assuming that the value for the variable *strName* is obtained from the user input, the XQuery would return the following, when the input provided by the user is *"john"*.

```
<user category="group1">
    <fname>john</fname>
    <status>good</status>
</user>
<user category="admin">
    <fname>john</fname>
    <status>good</status>
</user>
```

As the input string from the user is not validated properly, an attacker can provide input in such a way that the query is manipulated for retrieving the complete set of users. By providing the input string *XXX' or '1'='1* for the variable *strName*, the attacker can make the XQuery to return a node-set of all the users. The malformed XQuery is:

```
for $x in doc('users.xml')/userlist/user
 where $x/fname='XXX' or '1'='1' return $x
```

The XQuery when executed retrieves the whole XML document and presents it to the attacker. It can be observed that absence of proper validation mechanism for the user input has resulted in an attack. This security weakness leading to the XQuery injection attack is referred using *CWE-652*[1] (Improper Neutralization of Data within XQuery Expressions ('XQuery injection')).

Different types of XML injection attacks suggested by OWASP (2015) are listed as follows:

- *Tautology attack:* This attack is carried out by appending an expression that always returns 'True' to the user input. E.g., *or '1'= '1, or 'a'= 'a*. The example discussed above is a tautology attack.
- *Meta Character injection attack:* This attack is carried out by inserting an XML meta character such as ',",⟨,⟩, or *&* in the user input. The CWE identifier assigned for this type of attack is *CWE-150*.

[1]`https://cwe.mitre.org/data/definitions/652.html`

- *Comment injection attack:* This attack is performed by inserting a comment character sequence ⟨*!--* in the user input *(CWE-151).*
- *CDATA section injection attack:* This attack is carried out using the CDATA section *(CWE-146).* CDATA sections are generally used for escaping blocks of text which would otherwise be recognized as markup.
- *Tag injection attack:* This attack is carried out by injecting a tag in the user input causing the structure of the XML to be modified and hence corrupting the database.
- *External entity injection attack:* This attack occurs when XML input containing a reference to an external entity is processed by a weakly configured XML parser (XXE 2016). The set of valid entities that can be processed by the XML parser can be extended by defining new entities. If the definition of an entity is a URI, it is called an external entity. If it is not configured properly, then external entities force the XML parser to access confidential resource such as a file placed in a remote system *(CWE-611).*

5.2 PROBLEM DESCRIPTION

Given a web application, the objective of this work is to develop a systematic approach that is capable of generating attack vectors for detecting XQuery injection vulnerabilities prevailing in web applications driven by NXDs, independent of the functionality and technology of the application. The attack vectors should be capable of detecting all types of XQuery injection attacks specified by the security consortium OWASP. The proposed approach should also allow extensibility to address new kinds of attacks when discovered.

5.3 PROPOSED APPROACH

The proposed approach named *XiAttackGen* (XQuery injection Attack Generator) identifies vulnerabilities prevailing in web applications by penetrating the application with different types of malicious input strings. To detect XQuery injection, the malicious input strings are provided in the form fields available in the web pages of the application. The form fields are probable points of injection and are identified using a crawler. The malicious strings are generated using an attack grammar which is constructed by taking into account the different types of XML injections specified by OWASP, and the new categories of attack vectors listed in Section 5.3.3. The malicious strings assist in exposing the vulnerabilities in the applications from the obtained responses. The vulnerabilities identified assist the developers in fixing the coding defects existing in the web application so that the application becomes resistant to XML/XQuery injection attacks. This section elaborates on the approach followed for identifying probable injection points, and generation of malicious strings for identifying the vulnerabilities.

5.3.1 Identification of Injection Points

The web application under test is crawled starting from the seed URL provided by the tester. Seed URL is the starting point from which the web application under test is crawled for discovering the probable points of injection. The crawler developed as part of the R&D project navigates through the web pages of the application, and identifies forms and form fields through which the user interacts with the application. The information related to the forms and form fields are stored in the database for further processing. These form fields are considered as the possible points of injection. These injectable points are populated with invalid input strings for identifying vulnerabilities.

5.3.2 Attack Generation Based on OWASP Guidelines

Attack vectors for identifying the vulnerabilities are generated by proposing a context-free grammar. The attack grammar is framed based on regular expressions representing each type of injection attack as detailed in Sections 5.1 and 5.3.3. In this work, a regular expression has been framed to generate malicious strings representing each of these types of attacks. By combining these regular expressions, the attack grammar has been formed. Hence, the proposed attack grammar enables the identification of all the above mentioned types of XML injection attacks. For example, the regular expression for generating tautology attack strings is: *(A-Za-z0-9)*$^{+}$*’ or ‘1’= ‘1’ or ‘(A-Za-z0-9)*$^{+}$. A sample string generated by this regular expression is *abc’ or ‘1’= ‘1’ or ‘xyz*. The regular expression for comment injection attack string is: *(A-Za-z0-9)*$^{+}$ $\langle$*!--*. A sample string generated by this regular expression is *abc123* $\langle$*!--*. Similarly, regular expressions can be formed for each type of attack string. A combination of these regular expressions is used to form a context free grammar.

The attack grammar is a context free grammar *G* defined by *(N, T, P, S)* where *N* is the set of non-terminals, *T* is the set of terminals, *P* is the set of productions, and *S* is the start symbol.

N=$\{$*S, TAUT, METACHAR, COMMENT, CDATA_END, CDATA_CHAR, TAG, EXT_ENTITY, W_CDATA, ALT_ENCODING, EVAL, STRING_SC, SPECIAL_CHAR, STRING*$\}$

T=$\{$*‘, ’, “, ”,* $\langle$ *,* $\rangle$ *, &, !, [, /, ?, ., -, =, ;, , , :, @,* $\{$, $\}$*, [,], (,), _, a, b, ..., z, A, B, ..., Z, 0, 1, ..., 9*$\}$

Productions *P* of the grammar *G* are formed as follows:

S $\rightarrow$ *TAUT* | *METACHAR* | *COMMENT* | *CDATA_END* | *CDATA_CHAR* | *TAG* | *EXT_ENTITY* | *ALT_ENCODING* | *EVAL*

TAUT $\rightarrow$ *STRING’ or ‘1’=‘1’ or ‘STRING* | *STRING’ or ‘a’=‘a’ or ‘STRING* | *STRING’ or 1=1 or ‘STRING* | *STRING” or ‘1’=‘1’ or “STRING* | *STRING” or ‘a’=‘a’ or “STRING* | *STRING” or 1=1 or “STRING*

METACHAR $\rightarrow$ *STRING” STRING* | *STRING“ STRING* | *STRING‘ STRING* | *STRING’ STRING* | *STRING* $\langle$ *STRING* | *STRING* $\rangle$ *STRING* | *& STRING COMMENT* $\rightarrow$ *STRING*$\langle$*!--* | *(: STRING :)*

CDATA_END $\rightarrow$ *STRING]]*$\rangle$

CDATA_CHAR $\rightarrow$ *<![CDATA[W_CDATA]]>*

W_CDATA $\rightarrow$ *TAUT* | *TAG*

TAG $\rightarrow$ *<script>STRING</script>*

STRING_SC $\rightarrow$ *STRING_SC* | *STRING SPECIAL_CHAR STRING* |ϵ

SPECIAL_CHAR $\rightarrow$ *SPECIAL_CHAR* | _ | ; | : | @ | . | , | - | [|] | | | / | ! |ϵ

ALT_ENCODING $\rightarrow$ {*convert:binary-to-string(xs:hexBinary ('NUMBER'))*}

EVAL $\rightarrow$ {*eval('NUMBER OP NUMBER')*}

NUMBER $\rightarrow$ *NUMBER* | *0* | *1* | *2* | .... | *8* | *9* |ϵ

OP $\rightarrow$ + | - | * | / | % |ϵ

STRING $\rightarrow$ *STRING* | *a* | *b* |...| *z* | *A* | *B* |...| *Z* | *0* | *1* |...| *9* | _ | ϵ

The possible points of injection (i.e., form fields) identified in the web pages are populated with these attack strings, and a request is submitted to the web server. Vulnerability would be reported in the case of receiving a successful response from the web server.

Proof of Consistency and Completeness: The attack grammar framed is a context free grammar G. Generally the consistency and completeness of a context free grammar is proved using mathematical induction (Gopalakrishnan 2006). However, this technique is most suitable for grammars containing recursive productions especially in the start production. As the attack grammar framed in Section 5.3.2 does not contain recursive productions except to generate alphanumeric strings, the proof of consistency and completeness for this grammar is trivial. Consistency ensures that all strings generated by the grammar belong to the language for which the grammar is intended to be framed. Completeness ensures that all strings that belong to the language are generated by the grammar.

Proof of consistency: Let L be the language of XQuery attack strings. The start production for the grammar *G* is

S $\rightarrow$ *TAUT* | *METACHAR* | *COMMENT* | *CDATA_END* | *CDATA_CHAR* | *TAG* | *EXT_ENTITY* | *ALT_ENCODING* | *EVAL*

Consistency of the grammar is ensured when no string generated by S violate the rules of being in the language L. A detailed explanation on consistency of the grammar is provided via one of the productions *S* $\rightarrow$ *TAUT*. Let L^T be a subset of L, and represents the language of tautology attack strings given by,

L^T = {*(STRING' | STRING") (or 1=1 | or 'a'='a'| or '1'='1'| or "a"="a") (or 'STRING | or "STRING*)^m, *where* $m \geq 0$},

The non-terminal *TAUT* can be rewritten as follows:

TAUT $\rightarrow$ *STRING' or '1'='1' TAUT_ADD* | *STRING' or 'a'='a' TAUT_ADD* | *STRING' or 1=1 TAUT_ADD* | *STRING" or '1'='1' TAUT_ADD* | *STRING" or 'a'='a' TAUT_ADD* | *STRING" or 1=1 TAUT_ADD,* and

TAUT_ADD $\rightarrow$ *or 'STRING TAUT_ADD* | *or "STRING TAUT_ADD* |ϵ

Based on the definition of the language L^T, if *TAUT* generates a string containing any one of the following expressions *'1'= '1'*, *"1"="1"*, *1=1*, *'a'= 'a'*, or *"a"="a"*, then consistency is guaranteed for the production *S* $\rightarrow$*TAUT*.

Let *W* be a word generated by the *TAUT* production. Say, *S* $\rightarrow$*TAUT* generates *W*. When *TAUT_ADD* $\rightarrow$ ϵ, then the generated word *W* contains any one of the following expressions *'1'= '1'*, *"1"="1"*, *1=1*, *'a'= 'a'*, or *"a"="a"* as substring. The generated string is an attack string that belong to the language L^T under the category of tautology attacks. By induction hypothesis, if *TAUT_ADD* generates any other terminals other than ϵ, then the generated word contains any one of the aforementioned expressions. Thus, all the words generated by the production *S →TAUT* belong to the language L^T. Similarly, consistency can be proved for a word *W* generated by any of the other non-terminals on the right-hand-side of S.

Proof of completeness: The proof for completeness should ensure that any string that belong to the language of attack strings L can be generated by the grammar G. Let *W* be a word which belongs to the language of XQuery attack strings L. Say, *W* also belongs to the language of tautology attack strings L^T, where $L^T \subseteq L$. All the words produced from the given set of productions *TAUT* and *TAUT_ADD*, contains *'1'='1'*, *"1"="1"*, *1=1*, *'a'='a'*, or *"a"="a"* as substring when $m = 0$, $m = k$, and $m = k+1$. Therefore, it is evident that the productions can generate all set of strings that belong to the language L^T. Thus, all tautology attack strings in the language L can be generated by the grammar G. Similarly, completeness can be proved for all strings that belong to any category of attacks in L, all strings of which are generated by the corresponding start production. Therefore, the given grammar is complete enough to generate all the strings that belong to the language L.

5.3.3 Identification of XQuery Injection Attacks not Listed in OWASP

Apart from the attacks specified in OWASP (2015), new kinds of attacks are always possible and an adversary can make use of the advancements offered by the query language to manipulate the database request for launching attacks. The vulnerabilities in the test applications are exploited to launch different kinds of XQuery injection attacks that are not specified in OWASP (2015). The extendability of the proposed attack grammar is evidenced from the three new categories of XQuery injection attacks identified as part of this work and are discussed in the following subsections.

5.3.3.1 Alternate Encoding Attack

In this attack, the malicious code is injected in encoded form (e.g., ASCII, hexadecimal, etc.) into the query. The malicious code is encoded by the attacker to defeat the defensive coding practices employed for blocking special characters such as ampersand (&), single quote ('), and less than symbol (<) in the user input. A conversion function such as convert (2016) is included in the user input to decode the encoded attack string. This attack is generally used in conjunction with other types of attacks. In other words, alternate encodings do not provide any unique way to attack an application; it is an attack that allows malicious users a way to avoid the detection and prevention functions existing in the application, and therefore allows exploitation of vulnerabilities that might not otherwise be exploitable (Halfond et al. 2006).

Example 1: *"xquery let $up: ="+ parameter + "return insert node (<post><from> User1 </from><data>{$up}</data></post>) as last into doc ('postsdb/posts.xml')/ posts_all"*

Attack String:

$parameter = convert : binary - to - string(xs : hexBinary(`3c212d2d'))$

The query generated by the application is:

xquery let $up: = convert:binary-to-string(xs:hexBinary('3c212d2d'))
return insert node (<post><from>User1</from><data>{$up}</data></post>)
as last into doc('postsdb/posts.xml')/posts_all

The string equivalent of the hexadecimal code *'3c212d2d'* is *<!--*, a comment tag in HTML. Therefore, the query will be evaluated as:

*insert node (<post><from>User1</from><data><**!--**</data></post>) as last into doc ('postsdb/posts.xml')/posts_all*

When the query is executed, a new record is inserted into the database with the comment tag stored as a string '&lt;!-- '. Whenever a web page makes use of this injected record, the HTML comment tag gets inserted into the HTML code of the web page, and the rest of the HTML code is commented out from the point of insertion, and thus an attack is made successfully.

Example 2: Assuming that the application blocks special characters like $<$, $>$, etc. in the user input, the application can still be exploited by using an encoded version of the characters. For the same query discussed above, a stored XSS attack is possible with the following attack string.

Attack String:

$parameter = convert : binary - to - string(xs : hexBinary(`3c7363726970743e616c6572742827787373327293c2f7363726970743e'))$

The query generated by the application is:

xquery let $up: = $convert : binary - to - string(xs : hexBinary(`3c7363726970743e616c6572742827787373327293c2f7363726970743e'))$

return insert node (<post><from>User1</from><data>{$up}</data> </post>) as last into doc ('postsdb/posts.xml')/posts_all

The query will be evaluated as:

insert node (<post><from>User1</from><data>

<script>alert('xss') </script>

</data></post>) as last into doc ('postsdb/posts.xml')/posts_all

Consequently, a new child node gets inserted into the XML document and whenever a web page makes use of this injected record, a JavaScript alert will be thrown on the web page. Thus, encoded version of the malicious characters can be used along with *convert* function of XQuery for causing attacks.

5.3.3.2 Injection Through Evaluation Function

The absence of proper sanitization mechanisms for validating the user input can be exploited by an attacker for creating attacks that affect the intended behavior of the application. The expected behavior can be modified due to execution of malicious expressions by using *eval (2016)* function.

Consider a web page in an application that allows a user to enter a string against the price of a product. As the input field accepts string values, an adversary can provide input with an arithmetic expression embedded within *eval* function of XQuery. Suppose, the application fails to validate the value of price (i.e., fails to check if the value of price is greater than zero) at the client-side, then the adversary can provide the expression in such a way that it evaluates to a negative value thus affecting the intended behavior of the application. In case, client-side validation is done but lacks server-side validation, then still there is a possibility to inject the *eval* function for parameter price via the HTTP request.

Example 3: *"xquery let $up: =" + parameter + "return insert node (<product><name>Item1 </name><price>{$up}</price></product>) as last into doc ('products.xml')/products_all"*

Attack String:

$parameter = eval('100 * -1')$

The query generated by the application is:

xquery let $up: = eval('100*−1') return insert node (<product><name>Item1</name> <price>{$up}</price></product>) as last into doc ('products.xml') /products_all

The function *eval* evaluates the given expression and the query becomes as follows:

*insert node (<product><name>Item1 </name><price>**−100**</price></product>) as last into doc('products.xml')/products_all*

When the query is executed, a new product gets inserted into the database with the price of the product as **−100** and thus an injection is made successfully. The *eval* function can also be used to evaluate an XQuery expression.

5.3.3.3 XQuery Comment Injection Attack

In this attack, an attacker injects XQuery comment symbol (‘(:’) into the query so as to ignore a part of the XQuery expression from being executed which in turn affects the intended behavior of the application.

Example 4: *“xquery for $x in doc(‘users.xml’)/users/user where $x/fname=‘”+ strName + “’ and $x/password =‘”+ strPassword + “’ and $x/year=”+ iYear + “ return $x”*

Attack String:

$fname = admin'$(:

$password$ =abc

$year$=1975 :)

The query generated by the application is:

xquery for $x in doc(‘users.xml’)/users/user

where $x/fname = ‘admin’ (:’ and $x/password=‘abc’ and $x/year = 1975 :)

return $x

The query will be evaluated as:

xquery for $res in doc(‘users.xml’)/users/user where $x/fname = ‘admin’

return $x

When the query is executed, the attacker gains access to the application with the privileges of an administrator without submitting valid credentials. Thus, an attacker can bypass the authentication page using XQuery comment injection.

Example 5: *“xquery for $res in doc(‘students.xml’)/students/student where $res/-total <”+ ulimit + “and $res/total >”+ llimit + “ return (replace value of node with ‘First Class’)”*

Attack String:

$ulimit$ = 20(:

$limit$ = :)

The query generated by the application is:

xquery for $res in doc('students.xml')/students/student

where $res/total <20 (: and $res/total >:)

return (replace value of node with 'First Class')

The query will be evaluated as:

xquery for $res in doc('students.xml')/students/student where $res/total <20

return (replace value of node with 'First Class')

When the query is executed, the students who have scored marks less than 20 will be assigned to "First Class". Thus, an untrusted user can violate the intended behavior of the application.

5.4 EXPERIMENTAL STUDY AND ANALYSIS

This section provides details regarding the testbed on top of which the proposed approach is implemented and the applications that are used for evaluation. Extensive experiments are carried out to substantiate the effectiveness of the proposed approach which are discussed in detail in this section.

5.4.1 Experimental Setup

The proposed approach *XiAttackGen*, for generation of attack vectors to identify XQuery injection vulnerabilities in native XML database-driven web applications, is embedded within the prototype XQueryFuzzer developed as part of the R&D project supported by MeitY, Government of India. XQueryFuzzer involves the following phases: Crawling, Learning, Attack Generation, and Discovery. Figure 5.1 represents the workflow of the prototype XQueryFuzzer, and the components of *XiAttackGen* embedded within XQueryFuzzer. *XiAttackGen* forms the attack generation module of XQueryFuzzer.

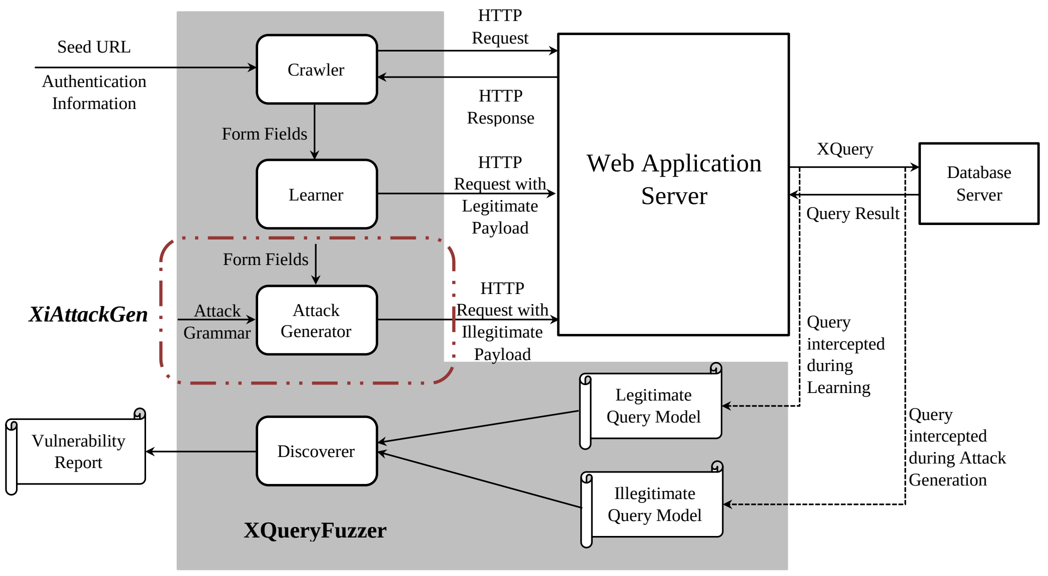

Figure 5.1: XQueryFuzzer architecture[2]

Crawling: The application under test is crawled starting from the seed URL provided by the tester. Using the seed URL as a starting point, the prototype scans all other web pages of the application to identify forms. The forms and form fields through which the user interacts with the application are identified and stored in the database. These form fields are considered as the points of injection.

Learning: The form fields identified during crawling phase are populated with valid inputs, and then the corresponding web form is submitted to the web application server using an HTTP request. The XQueries generated by the web application server on processing the request are sent to the XML database server for execution. The queries executed successfully are intercepted, and utilized for constructing legitimate query models. The queries are intercepted by modifying the database driver files specific to the native XML database used by the web application under test.

Attack Generation: The injectable points are populated with attack strings that are generated based on the attack grammar, and the corresponding web form is submitted to the application. The queries that are processed successfully by the server on submission of the attack strings are intercepted, and then used for constructing illegitimate query models.

[2]The prototype is developed as part of R&D project supported by MeitY, Government of India.

Table 5.1: Test application details

Test Application	Description	Web Technology	# Hotspots
BookStore	Online bookstore	JSP	71
Classifieds	Online management system for classifieds	JSP	34
Employee Directory	Online employee directory	JSP	23
Events	Event tracking system	JSP	31

Discovery: The illegitimate query models are compared against their respective legitimate query models for identifying vulnerabilities. Vulnerabilities are reported when there is a mismatch between the generated query models.

The details regarding construction and comparison of legitimate and illegitimate query models can be found in Palsetia et al. (2016).

5.4.2 Test Applications

The test suite developed by Halfond and Orso (2005) is used for testing the proposed approach. The applications BookStore, Classifieds, Employee Directory, and Events are used for evaluation. These applications use a relational database at the backend for storing the data. For evaluation of the proposed approach, the database of these applications is modified to BaseX (NXD), and SQL queries in the application are replaced with XQueries. The test applications given in Table 5.1 lists the name of the application, description, technology involved, and the number of locations that issue XQueries (hotspots).

5.4.3 Experimental Results and Discussions

The effectiveness of the proposed approach is evaluated along two dimensions: (1) experimental effectiveness, and (2) comparison with Zed Attack Proxy (ZAP) (ZAP 2016), an open-source vulnerability scanner. The evaluation results are discussed below.

Table 5.2 describes the number of attack requests submitted to the test application, number of successful attacks, number of forms existing in the application, number of vulnerable forms existing in the application, and number of vulnerable forms detected by the prototype. The table shows that for applications BookStore and Classifieds, two

Table 5.2: XQuery injection vulnerabilties detected

Test Application	#Attack Requests	#Successful Attacks	#Forms	#Vulnerable forms	
				Existing	Detected
BookStore	726	235	32	19	17
Classifieds	528	125	20	14	12
Employee Directory	286	111	9	7	7
Events	396	157	12	10	10
Total	1936	628	73	50	46

forms are missed from being identified as vulnerable. The forms are missed because of the validation placed on few parameters on the server-side. As the crawler component of the prototype is not aware of the context of the application, valid values could not be placed for the parameters that have restrictions on it, and hence both legitimate and illegitimate requests fail for both the cases. All the existing vulnerable forms are identified by the prototype for applications Events and Employee Directory.

Table 5.3 lists the number of vulnerabilities existing in the test application, the number of vulnerabilities detected by the prototype, the number of true positives, false positives and false negatives. On analyzing the results, it is found that the prototype detects maximum number of vulnerabilities present in the test applications with few false positives. The overall true positive rate of the proposed approach is found to be 93.27%.

False Positives and False Negatives: Table 5.4 gives the number of false positives and false negatives reported by the prototype. The prototype resulted in two false positives and four false negatives. False positives are reported in applications BookStore and Employee Directory. The reason behind this is the number of queries executed during legitimate and illegitimate requests accidentally matches, even though the illegitimate request was not processed successfully. False positives can be eliminated by comparing the HTTP responses obtained for legitimate and illegitimate requests of the application. The reason behind false negatives is explained below. In the case of application BookStore, web page *ShoppingCartRecord.jsp* is not visited by the crawler as the parent page *ShoppingCart.jsp* did not have any items that provide hyperlinks to ShoppingCartRecord page during the visit. Vulnerabilities in three other pages namely,

Table 5.3: Number of vulnerabilities existing in test applications and detected by the proposed approach

	BookStore				**Classifieds**				**Emp. Directory**				**Events**			
Type of injection attack	E	D	FP	FN	E	D	FP	FN	E	D	FP	FN	E	D	FP	FN
Tautology injection	21	20	1	2	15	13	0	2	7	7	0	0	10	10	0	0
Comment injection	14	15	2	1	9	8	0	1	3	5	2	0	3	3	0	0
Meta character ⟩ injection	21	19	0	2	15	13	0	2	7	7	0	0	10	10	0	0
Meta character & injection	0	3	3	0	0	0	0	0	0	2	2	0	0	0	0	0
Meta character ⟨ injection	14	15	2	1	9	8	0	1	3	5	2	0	3	3	0	0
Tag injection with CDATA	5	5	1	1	2	1	0	1	4	4	0	0	7	7	0	0
CDATA end character sequence injection]]⟩	16	14	0	2	12	10	0	2	7	7	0	0	10	10	0	0
Tag injection	21	19	0	2	15	13	0	2	7	7	0	0	10	10	0	0
Special character (single/-double quotes) injection	12	11	1	2	7	5	0	2	8	8	0	0	14	14	0	0
Alternate encoding injection	16	14	0	2	12	10	0	2	7	7	0	0	10	10	0	0
Evaluation function injection	16	15	1	2	10	8	0	2	7	7	0	0	10	10	0	0
External entity injection	0	3	3	0	0	0	0	0	0	2	2	0	0	0	0	0

E – Vulnerabilities Existing, D – Vulnerabilities Detected, FP – False Positives, FN – False Negatives

Table 5.4: Effectiveness of the proposed approach

Test Application	#Vulnerablilities		# TP	# FP	# FN	True Positive Rate(%)
	Existing	Detected				
BookStore	156	153	139	14	17	89.10
Classifieds	106	89	89	0	17	83.96
Employee Directory	60	68	60	8	0	100.00
Events	87	87	87	0	0	100.00
Total	409	397	375	22	34	93.27

TP – True Positives, FP – False Positives, FN – False Negatives

Table 5.5: Comparison of the proposed approach with ZAP

Test Application	#Vulnerable Forms		
	Existing	Detected by Prototype	Detected by ZAP
BookStore	19	17	11
Classifieds	14	12	9
Employee Directory	7	7	6
Events	10	10	6

OrdersRecord.jsp, BookMaint.jsp and *BookDetail.jsp*, are missed from being identified since they have constraints on one of the text fields. Similarly, application Classifieds has one page *MyAdRecord.jsp* which has a constraint on one of the text fields, and hence is missed from being identified. The false negatives can be eliminated by using a context-aware crawler, and false positives can be eliminated by comparing the HTTP response of the application. Thus, the false positives and false negatives obtained are due to the crawler and discoverer components of the prototype and are not due to the proposed grammar.

Comparison with ZAP: Table 5.5 gives the results of the proposed approach as compared to the open-source penetration testing tool ZAP (2016) of OWASP. The table shows the number of vulnerable parameters identified by the prototype and the penetration testing tool, ZAP. Column 3 lists the number of parameters that are vulnerable to XQuery injection, whereas column 4 lists the number of parameters that are vulnerable to SQL injection. Even though ZAP identifies SQLIVs, the parameters that are placed in the SQL query and XQuery for exploiting the application are the same. From the table, it can be inferred that the developed prototype identifies more vulnerabilities compared to ZAP.

5.4.4 Advantages and Limitations

The advantage of the proposed approach is that it identifies known patterns of attacks, but in case a new attack is to be considered, a regular expression can be framed representing the attack and added as a production to the attack grammar. Hence, this approach is flexible and easily extendable for new attack vectors. The extensions that can be made to the proposed approach is updating the attack grammar with additional productions, when new types of vulnerabilities are discovered.

Other types of vulnerabilities such as those leading to blind XPath injection attack are not addressed by the proposed attack grammar. Different from other types of XML injection attacks, blind XPath injection is a special kind of injection which focuses on extraction of information about data stored in the XML document by asking a series of true/false questions (i.e., booleanized queries) (Klein 2005). The types of injection addressed by the proposed approach are straightforward, and either extract a piece of information or inject untrusted information into the database, whereas blind XPath injection attack employs two methods such as XPath crawling and booleanization for inferring information from the XML document. Each injection query extracts a single bit of information from the XML document, and the whole of the XML document is inferred from unlimited number of queries, which is conceptually different from the proposed approach, and hence it is not addressed as part of this work.

5.5 SUMMARY

XML injection has become a critical vulnerability with the increased use of XML databases by web applications. XML injection vulnerabilities need to be detected and corrected so that the web application is secure against various types of XQuery injection attacks. Hence, an approach for launching attack vectors for identification of XQuery injection vulnerabilities has been developed. The approach generates malicious strings using an attack grammar, which are filled in the possible points of injection. The attack grammar is capable of generating various types of XML injection strings specified in OWASP guidelines and the new categories of attack vectors identified as part of this work. The possible points of injection are identified by comparing the queries generated during normal and attack executions. The proposed approach has been exhaustively tested on customized benchmark web applications, and is found to work effectively with minimum number of false positives and false negatives. Additionally, this chapter discussed three new categories of attacks identified as part of this work.

CHAPTER 6

CONCLUSIONS AND FUTURE SCOPE

The security of web applications is affected due to vulnerabilities that are incredibly prevalent, and these vulnerabilities allow unrestricted access to confidential and sensitive data for malicious users on exploitation. In order to counteract this trend, there is a growing need to create automated approaches that are capable of finding the security vulnerabilities in web applications. Black-box approach for detection of vulnerabilities in web applications is a promising avenue that offers the flexibility to identify and rectify the vulnerabilities irrespective of the functionality and technology of the application. The appropriate generation of attack vectors based on the behavior of the application during black-box detection assists in effective detection of the vulnerabilities. Hence, this thesis presents a behavior-based approach to generate attack vectors for the detection of vulnerabilities in web applications independent of the functionality of the applications. Logic and XQuery injection vulnerabilities have become critical vulnerabilities with the increased use of web applications, and hence the first part of the work focused on generation of attack vectors to detect logic vulnerabilities, and the second part was on generation of attack vectors to detect XQuery injection vulnerabilities.

6.1 SUMMARY OF THE CONTRIBUTIONS

The proposed approach for identifying logic vulnerabilities employs black-box analysis for inferring the intended behavior of the application to launch attack vectors. The proposed approach models the web application as an annotated FSM by using the execution traces, and extracts constraints from the model to reflect the behavior of the application. The constraints provide information about the user-input parameters that are constrained at the client-side, parameters flowing between different web pages, web pages that require session variables to be defined, web pages accessible by privileged users, and sequence of operation of the web pages. These constraints reflect the data flow and control flow of the application. The extracted constraints are violated to generate attack vectors for the detection of vulnerabilities. The approach has been exhaustively tested on vulnerable web applications, and it provides promising results. The proposed approach is also compared with the existing literature, and the results substantiate the effectiveness of the approach. The proposed FSM model is flexible enough to identify vulnerabilities leading to a new kind of attack, namely session puzzling. It also offers the flexibility to identify EAR vulnerabilities. Hence, this systematic approach for identifying logic vulnerabilities can become an essential component to ensure the security of the applications against logic attacks.

The approach defined for addressing XQuery injection vulnerabilities proposes an attack grammar for the generation of attack vectors to detect the vulnerabilities in native XML database-driven web applications. The proposed grammar is tested on vulnerable web applications customized for the purpose, and analysis of the results shows that the approach is capable of identifying most of the vulnerabilities in the web applications. Three new categories of XQuery injection attack vectors namely *alternate encoding, evaluation function,* and *XQuery comment injection attacks* not listed in OWASP are discovered as part of this work. Additionally, the consistency and completeness of the attack grammar is proved to ensure that the grammar is capable of generating all kinds of attack strings specific to the OWASP guidelines. The grammar assists in identifying known attacks, and if a new attack is to be considered, a regular expression representing the attack is to be framed and added as a production to the attack grammar. Thus, this approach is flexible and easily extendable for new attack vectors. This systematic approach for identifying XQuery injection vulnerabilities can become an essential component for XML database-driven web applications to ensure the security of the applications against injection attacks.

6.2 FUTURE SCOPE

With emerging technologies, web applications are evolving fast and therefore creates ever growing opportunities for advanced features in the field of web application security. This section outlines the future research directions related to the proposed work, and points out the pioneering work in the domain of securing web applications from attacks.

- Firstly, the correctness and completeness of the results obtained during detection of logic vulnerabilities are dependent on the user-generated traces of the application. As a future work, the manual trace generation involved for modeling the intended behavior of the application can be automated using a simulated/emulated crawler so that all possible web pages in the application could be explored.
- Secondly, the proposed approach takes only session variables into account for detecting access-control vulnerabilities. As a consequence, vulnerabilities arising due to the access-control policies implemented through the database of the application are missed from being identified. As a future work, it would be interesting to extract the access constraints placed on the database of the application in addition to the session variables to enhance the detection of access-control vulnerabilities.
- Thirdly, web applications today are constructed by integrating with third-party web services, which increases the complexity of the applications. Logic vulnerabilities arising from the integrating functionalities could be identified by understanding the specification of the application under multi-party integration scenarios. As a future direction, it would be interesting and promising to extend the proposed approach to identify vulnerabilities arising due to integration among multiple web applications.
- Fourthly, the recent research works addressing injection and logic vulnerabilities concentrate on fixing the flaws automatically after detection, and alarming the programmer regarding the possible vulnerabilities during construction of the application itself. Hence, it would be helpful to propose an approach for fixing the discovered flaws.

- Finally, recent works are pioneering towards detection and prevention of vulnerabilities and attacks using machine learning approaches. As a future direction, it would be promising to train the constructed FSM Model using machine learning algorithms to improve the precision of the system.

To conclude, this work proposed a behavior-based approach for generation of attack vectors to identify three types of logic vulnerabilities. Additionally, a context-free grammar is proposed for generation of attack vectors to identify different types of XQuery injection vulnerabilities in web applications driven by NXDs.

BIBLIOGRAPHY

Acunetix. "Acunetix Vulnerability Scanner." `https://www.acunetix.com/vulnerability-scanner/` (June 6, 2015).

Acunetix (2010). "AT&T Website." `https://www.acunetix.com/blog/articles/analysis-php-attack-apple-information-disclosure/` (July 26, 2017).

Al-Hamdani, W. A. (2010). "XML security in healthcare web systems." In *2010 Information Security Curriculum Development Conference*, InfoSecCD '10, ACM, New York, USA, 80–93.

Alkhalaf, M., Choudhary, S. R., Fazzini, M., Bultan, T., Orso, A. and Kruegel, C. (2012). "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*, ISSTA 2012, ACM, New York, USA, 56–66.

Antunes, N., Laranjeiro, N., Vieira, M. and Madeira, H. (2009). "Effective detection of SQL/XPath injection vulnerabilities in web services." In *2009 IEEE International Conference on Services Computing*, Bangalore, India, 260–267.

Antunes, N. and Vieira, M. (2011). "Enhancing penetration testing with attack signatures and interface monitoring for the detection of injection vulnerabilities in web services." In *2011 IEEE International Conference on Services Computing*, Washington, USA, 104–111.

Antunes, N. and Vieira, M. (2012). "Defending against web application vulnerabilities." *Computer*, 45(2), 66–72.

Arachni (2016). “Arachni - web application security scanner framework.” `http://www.arachni-scanner.com/` (Apr. 5, 2016).

Asmawi, A., Affendey, L. S., Udzir, N. I. and Mahmod, R. (2012). “Model-based system architecture for preventing XPath injection in database-centric web services environment.” In *2012 7th International Conference on Computing and Convergence Technology (ICCCT)*, Seoul, Rep. of Korea, 621–625.

Auger, R. (2010). “XQuery injection.” `http://projects.webappsec.org/w/page/13247006/XQueryInjection` (Apr. 5, 2016).

Balduzzi, M., Gimenez, C. T., Balzarotti, D. and Kirda, E. (2011). “Automated discovery of parameter pollution vulnerabilities in web applications.” In *Proceedings of the 18th Network and Distributed System Security Symposium*, NDSS’11, San Diego, USA.

Balzarotti, D., Cova, M., Felmetsger, V., Jovanovic, N., Kirda, E., Kruegel, C. and Vigna, G. (2008). “Saner: Composing static and dynamic analysis to validate sanitization in web applications.” In *2008 IEEE Symposium on Security and Privacy*, Oakland, USA, 387–401.

Balzarotti, D., Cova, M., Felmetsger, V. V. and Vigna, G. (2007). “Multi-module vulnerability analysis of web-based applications.” In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS ’07, ACM, New York, USA, 25–35.

Barth, A., Jackson, C. and Mitchell, J. C. (2008). “Robust defenses for cross-site request forgery.” In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS ’08, ACM, New York, USA, 75–88.

BaseX. “BaseX-the XML database.” `http://basex.org/` (Apr. 5, 2016).

Bau, J., Bursztein, E., Gupta, D. and Mitchell, J. (2010). “State of the art: Automated black-box web application vulnerability testing.” In *2010 IEEE Symposium on Security and Privacy*, Oakland, USA, 332–345.

Bau, J., Wang, F., Bursztein, E., Mutchler, P. and Mitchell, J. C. (2012). "Vulnerability factors in new web applications: Audit tools, developer selection & languages." Technical report, Stanford University.

Bisht, P., Hinrichs, T., Skrupsky, N., Bobrowicz, R. and Venkatakrishnan, V. N. (2010a). "Notamper: Automatic blackbox detection of parameter tampering opportunities in web applications." In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, ACM, New York, USA, 607–618.

Bisht, P., Hinrichs, T., Skrupsky, N. and Venkatakrishnan, V. N. (2011). "Waptec: Whitebox analysis of web applications for parameter tampering exploit construction." In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, ACM, New York, USA, 575–586.

Bisht, P., Madhusudan, P. and Venkatakrishnan, V. N. (2010b). "Candid: Dynamic candidate evaluations for automatic prevention of SQL injection attacks." *ACM Transactions on Information and System Security*, 13(2), 14:1–14:39.

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

Bourret, R. (2009). "Going native: Use cases for native XML databases." `http://www.rpbourret.com/xml/UseCases.htm` (Apr. 5, 2016).

Boyd, S. W. and Keromytis, A. D. (2004). "Sqlrand: Preventing SQL injection attacks." In *Applied Cryptography and Network Security: Second International Conference, ACNS 2004, Yellow Mountain, China, June 8-11, 2004. Proceedings*, Springer Berlin Heidelberg, 292–302.

Braun, B., Kucher, S., Johns, M. and Posegga, J. (2012). "A user-level authentication scheme to mitigate web session-based vulnerabilities." In *Trust, Privacy and Security in Digital Business*, volume 7449 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 17–29.

Bravenboer, M., Dolstra, E. and Visser, E. (2010). “Preventing injection attacks with syntax embeddings.” *Science of Computer Programming*, 75(7), 473–495.

Brijesh Deb, S. G. B. and Bharti, S. (2007). “Rich Internet Applications RIA: Opportunities and challenges for enterprises.” `http://www.infosys.com/IT-services/application-services/white-papers/Documents/rich-internet-applications.pdf`).

Buehrer, G., Weide, B. W. and Sivilotti, P. A. G. (2005). “Using parse tree validation to prevent SQL injection attacks.” In *Proceedings of the 5th International Workshop on Software Engineering and Middleware*, SEM ’05, ACM, New York, USA, 106–113.

Cao, Y., Yegneswaran, V., Porras, P. and Chen, Y. (2011). “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*, CCS ’11, ACM, New York, USA, 745–748.

Ceccato, M., Nguyen, C. D., Appelt, D. and Briand, L. C. (2016). “SOFIA: An automated security oracle for black-box testing of sql-injection vulnerabilities.” In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, ACM, New York, USA, 167–177.

Chan, G.-Y., Lee, C.-S. and Heng, S.-H. (2013). “Discovering fuzzy association rule patterns and increasing sensitivity analysis of XML-related attacks.” *Journal of Network and Computer Applications*, 36(2), 829–842.

Chaudhri, A., Zicari, R. and Rashid, A. (2003). *XML Data Management: Native XML and XML Enabled DataBase Systems*, Addison-Wesley Longman Publishing Co., Inc., Boston, USA.

Chaudhuri, A. and Foster, J. S. (2010). “Symbolic security analysis of ruby-on-rails web applications.” In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS ’10, ACM, New York, USA, 585–594.

Chen, S. (2011). “Session puzzles - indirect application attack vectors.” Technical report, Ernst & Young.

Chong, S., Liu, J., Myers, A. C., Qi, X., Vikram, K., Zheng, L. and Zheng, X. (2007a). "Secure web applications via automatic partitioning." In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, ACM, New York, USA, 31–44.

Chong, S., Vikram, K. and Myers, A. C. (2007b). "SIF: Enforcing confidentiality and integrity in web applications." In *Proceedings of the 16th USENIX Security Symposium*, SS'07, USENIX Association, Berkeley, USA, 1:1–1:16.

Ciampa, A., Visaggio, C. A. and Di Penta, M. (2010). "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*, SESS '10, ACM, New York, USA, 43–49.

convert (2016). "Conversion module." `http://docs.basex.org/wiki/Conversion_Module` (Apr. 5, 2016).

Corcoran, B. J., Swamy, N. and Hicks, M. (2009). "Cross-tier, label-based security enforcement for web applications." In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, ACM, New York, USA, 269–282.

Cova, M., Balzarotti, D., Felmetsger, V. and Vigna, G. (2007a). "Swaddler: An approach for the anomaly-based detection of state violations in web applications." In *Recent Advances in Intrusion Detection*, volume 4637 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 63–86.

Cova, M., Felmetsger, V. and Vigna, G. (2007b). "Vulnerability analysis of web-based applications." In *Test and Analysis of Web Services*, Springer Berlin Heidelberg, 363–394.

CSRF. "Cross-site request forgery (CSRF) prevention cheat sheet." `https://www.owasp.org/index.php/Cross-Site_Request_Forgery_%28CSRF%29_Prevention_Cheat_Sheet` (Jan. 29, 2017).

Daitch, H. (May 3, 2017). "Sophisticated gmail phishing scam targets users." `https://www.identityforce.com/blog/sophisticated-gmail-phishing-scam` (July 26, 2017).

Dalton, M., Kozyrakis, C. and Zeldovich, N. (2009). "Nemesis: Preventing authentication & access control vulnerabilities in web applications." In *Proceedings of the 18th USENIX Security Symposium*, SSYM'09, USENIX Association, Berkeley, USA, 267–282.

de la Torre, I., Díaz, F. J., Antón, M., Díez, J. F., Sainz, B., López, M., Hornero, R. and López, M. I. (2011). "Choosing the most efficient database for a web-based system to store and exchange ophthalmologic health records." *Journal of Medical Systems*, 35(6), 1455–1464.

Deepa, G. and Thilagam, P. S. (2016). "Securing web applications from injection and logic vulnerabilities: Approaches and challenges." *Information and Software Technology*, 74, 160–180.

Di Lucca, G., Fasolino, A., Mastoianni, M. and Tramontana, P. (2004). "Identifying cross site scripting vulnerabilities in web applications." In *Sixth IEEE International Workshop on Web Site Evolution (WSE'04)*, Chicago, USA, 71–80.

Dominos (2012). "Dominos' India website hacked, customer info leaked." `http://www.rediff.com/business/report/dominos-india-website-hacked-customer-info-leaked/20120911.htm` (July 26, 2017).

Doupé, A., Boe, B., Kruegel, C. and Vigna, G. (2011). "Fear the EAR: Discovering and mitigating execution after redirect vulnerabilities." In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, ACM, New York, USA, 251–262.

Doupé, A., Cova, M. and Vigna, G. (2010). "Why johnny cant pentest: An analysis of black-box web vulnerability scanners." In *Detection of Intrusions and Malware,* *and Vulnerability Assessment*, volume 6201 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 111–131.

Doupé, A., Cui, W., Jakubowski, M. H., Peinado, M., Kruegel, C. and Vigna, G. (2013). "dedacota: Toward preventing server-side XSS via automatic code and data separation." In *Proceedings of the 2013 ACM SIGSAC Conference on Computer &; Communications Security*, CCS '13, ACM, New York, USA, 1205–1216.

Duchene, F., Rawat, S., Richier, J.-L. and Groz, R. (2014). "Kameleonfuzz: Evolutionary fuzzing for black-box XSS detection." In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, CODASPY '14, ACM, New York, USA, 37–48.

eval (2016). "XQuery module." `http://docs.basex.org/wiki/XQuery_Module` (Apr. 5, 2016).

Evans, Z. and Shahriar, H. (2014). "Web session security: Attack and defense techniques." *Case Studies in Secure Computing: Achievements and Trends*, 389.

eXistDB. "eXistDB - the open source native XML database." `http://exist-db.org/exist/apps/homepage/index.html` (Apr. 5, 2016).

Felmetsger, V., Cavedon, L., Kruegel, C. and Vigna, G. (2010). "Toward automated detection of logic vulnerabilities in web applications." In *Proceedings of the 19th USENIX Conference on Security*, USENIX Security'10, USENIX Association, Berkeley, USA, 10–10.

Fonseca, J., Vieira, M. and Madeira, H. (2007). "Testing and comparing web vulnerability scanning tools for SQL injection and XSS attacks." In *13th Pacific Rim International Symposium on Dependable Computing, PRDC 2007*, Melbourne, Australia, 365–372.

Forbes, T. (2014). "Exploiting XPath injection vulnerabilities with XCat." `http://tomforb.es/exploiting-xpath-injection-vulnerabilities-with-xcat-1` (Apr. 5, 2016).

Garcia-Alfaro, J. and Navarro-Arribas, G. (2008). "A survey on detection techniques to prevent cross-site scripting attacks on current web applications." In *Critical Information Infrastructures Security*, volume 5141 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 287–298.

Garcia-Alfaro, J. and Navarro-Arribas, G. (2009). "A survey on cross-site scripting attacks." *arXiv preprint arXiv:0905.4850*.

Gopalakrishnan, G. (2006). *Computation Engineering-Applied Automata Theory and Logic*, Springer, USA.

Gordeychik, S. (2008). "Web application security statistics." `http://projects.webappsec.org/w/page/13246989/WebApplicationSecurityStatistics` (June 6, 2015).

Grabowski, R., Hofmann, M. and Li, K. (2012). "Type-based enforcement of secure programming guidelines code injection prevention at SAP." In *Formal Aspects of Security and Trust*, volume 7140 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 182–197.

Groppe, J. and Groppe, S. (2008). "Filtering unsatisfiable XPath queries." *Data & Knowledge Engineering*, 64(1), 134–169.

Halfond, W., Orso, A. and Manolios, P. (2008). "WASP: Protecting web applications using positive tainting and syntax-aware evaluation." *IEEE Transactions on Software Engineering*, 34(1), 65–81.

Halfond, W., Viegas, J. and Orso, A. (2006). "A classification of SQL-injection attacks and countermeasures." In *Proceedings of the IEEE International Symposium on Secure Software Engineering*, Washington, USA, 65–81.

Halfond, W. G. J. and Orso, A. (2005). "Amnesia: Analysis and monitoring for neutralizing SQL-injection attacks." In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, ACM, New York, USA, 174–183.

Hein, D. and Saiedian, H. (2009). "Secure software engineering: Learning from the past to address future challenges." *Information Security Journal: A Global Perspective*, 18(1), 8–25.

Hirsch, F. (2002). "Getting started with XML security." `http://www.sitepoint.com/getting-started-xml-security/` (Apr. 5, 2016).

Howard, M., LeBlanc, D. and Viega, J. (2009). *24 Deadly Sins of Software Security: Programming Flaws and How to Fix Them*, McGraw-Hill, Inc., USA.

Huang, L.-S., Moshchuk, A., Wang, H. J., Schechter, S. and Jackson, C. (2012). "Clickjacking: Attacks and defenses." In *Proceedings of the 21st USENIX Security Symposium*, Security'12, USENIX Association, Berkeley, USA, 22–22.

Huang, Y. (2003). "Safeguarding a native XML database system." `https://www.cs.auckland.ac.nz/courses/compsci725s2c/archive/termpapers/725huang.pdf` (Apr. 5, 2016).

Huang, Y.-W., Tsai, C.-H., Lin, T.-P., Huang, S.-K., Lee, D. and Kuo, S.-Y. (2005). "A testing framework for web application security assessment." *Computer Networks*, 48(5), 739–761.

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

Hydara, I., Sultan, A. B. M., Zulzalil, H. and Admodisastro, N. (2015). "Current state of research on cross-site scripting (XSS) a systematic literature review." *Information and Software Technology*, 58, 170–186.

Igure, V. and Williams, R. (2008). "Taxonomies of attacks and vulnerabilities in computer systems." *IEEE Communications Surveys & Tutorials*, 10(1), 6–19.

ITRC (2015). "Identity theft resource center breach report hits record high in 2014." `http://www.idtheftcenter.org/ITRC-Surveys-Studies/2014databreaches.html` (June 6, 2015).

Jan, S., Nguyen, C. D., Arcuri, A. and Briand, L. (2017). "A search-based testing approach for XML injection vulnerabilities in web applications." In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, Tokyo, Japan, 356–366.

Jan, S., Nguyen, C. D. and Briand, L. C. (2016). "Automated and effective testing of web services for XML injection attacks." In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, ACM, New York, USA, 12–23.

Jang, Y.-S. and Choi, J.-Y. (2014). "Detecting SQL injection attacks using query result size." *Computers & Security*, 44, 104–118.

Jia, L., Vaughan, J. A., Mazurak, K., Zhao, J., Zarko, L., Schorr, J. and Zdancewic, S. (2008). "Aura: A programming language for authorization and audit." In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, ACM, New York, USA, 27–38.

Jim, T., Swamy, N. and Hicks, M. (2007). "Defeating script injection attacks with browser-enforced embedded policies." In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, ACM, New York, USA, 601–610.

Johns, M. (2006). "Sessionsafe: Implementing XSS immune session handling." In *Computer Security ESORICS 2006*, volume 4189 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 444–460.

Johns, M., Beyerlein, C., Giesecke, R. and Posegga, J. (2010). "Secure code generation for web applications." In *Engineering Secure Software and Systems*, volume 5965 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 96–113.

Johns, M., Braun, B., Schrank, M. and Posegga, J. (2011). “Reliable protection against session fixation attacks.” In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC ’11, ACM, New York, USA, 1531–1537.

Johns, M., Engelmann, B. and Posegga, J. (2008). “XSSDS: Server-side detection of cross-site scripting attacks.” In *Annual Computer Security Applications Conference, ACSAC 2008*, Anaheim, USA, 335–344.

Johns, M. and Lekies, S. (2013). “Tamper-resistant likejacking protection.” In *Research in Attacks, Intrusions, and Defenses*, volume 8145 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 265–285.

Jovanovic, N., Kirda, E. and Kruegel, C. (2006a). “Preventing cross site request forgery attacks.” In *Securecomm and Workshops*, 1–10.

Jovanovic, N., Kruegel, C. and Kirda, E. (2006b). “Pixy: a static analysis tool for detecting web application vulnerabilities.” In *2006 IEEE Symposium on Security and Privacy*, Oakland, USA, 6 pp.–263.

Jovanovic, N., Kruegel, C. and Kirda, E. (2006c). “Precise alias analysis for static detection of web application vulnerabilities.” In *Proceedings of the 2006 Workshop on Programming Languages and Analysis for Security*, PLAS ’06, ACM, New York, USA, 27–36.

Juillerat, N. (2007). “Enforcing code security in database web applications using libraries and object models.” In *Proceedings of the 2007 Symposium on Library-Centric Software Design*, LCSD ’07, ACM, New York, USA, 31–41.

Kals, S., Kirda, E., Kruegel, C. and Jovanovic, N. (2006). “Secubat: A web vulnerability scanner.” In *Proceedings of the 15th International Conference on World Wide Web*, WWW ’06, ACM, New York, USA, 247–256.

Kar, D., Panigrahi, S. and Sundararajan, S. (2016). “Sqligot: Detecting SQL injection attacks using graph of tokens and SVM.” *Computers & Security*, 60, 206–225.

Keizer, G. (2012). "Yahoo fixes password-pilfering bug, explains who's at risk." `http://www.computerworld.com/article/2505836/cybercrime-hacking/yahoo-fixes-password-pilfering-bug--explains-who-s-at-risk.html` (July 26, 2017).

Kirda, E., Jovanovic, N., Kruegel, C. and Vigna, G. (2009). "Client-side cross-site scripting protection." *Computers & Security*, 28(7), 592–604.

Kirda, E., Kruegel, C., Vigna, G. and Jovanovic, N. (2006). "Noxes: A client-side solution for mitigating cross-site scripting attacks." In *Proceedings of the 2006 ACM Symposium on Applied Computing*, SAC '06, ACM, New York, USA, 330–337.

Klein, A. (2005). "Blind XPath injection." Technical report, Watchfire Corporation, Waltham, USA.

Klein, A. (2007). "DOM based cross site scripting or XSS of the third kind." `http://www.webappsec.org/projects/articles/071105.shtml`).

Kosuga, Y., Kernel, K., Hanaoka, M., Hishiyama, M. and Takahama, Y. (2007). "Sania: Syntactic and semantic analysis for automated testing against SQL injection." In *Twenty-Third Annual Computer Security Applications Conference, ACSAC 2007*, Florida, USA, 107–117.

Krishnamurthy, A., Mettler, A. and Wagner, D. (2010). "Fine-grained privilege separation for web applications." In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, ACM, New York, USA, 551–560.

Lampesberger, H. (2013). "A grammatical inference approach to language-based anomaly detection in XML." In *2013 International Conference on Availability, Reliability and Security*, Regensburg, Germany, 685–693.

Laranjeiro, N., Vieira, M. and Madeira, H. (2009). "Protecting database centric web services against SQL/XPath injection attacks." In *Proceedings of the 20th International Conference on Database and Expert Systems Applications, DEXA 2009*, Springer Berlin Heidelberg, 271–278.

Lee, I., Jeong, S., Yeo, S. and Moon, J. (2012). "A novel method for SQL injection attack detection based on removing SQL query attribute values." *Mathematical and Computer Modelling*, 55(12), 58–68.

Lee, K. K.-Y., Tang, W.-C. and Choi, K.-S. (2013). "Alternatives to relational database: Comparison of NoSQL and XML approaches for clinical data storage." *Computer Methods and Programs in Biomedicine*, 110(1), 99–109.

Lekies, S., Stock, B. and Johns, M. (2013). "25 million flows later: Large-scale detection of DOM-based XSS." In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, ACM, New York, USA, 1193–1204.

Lekies, S., Tighzert, W. and Johns, M. (2012). "Towards stateless, client-side driven cross-site request forgery protection for web applications." In *Sicherheit 2012*, Lecture Notes in Informatics.

Li, X., Si, X. and Xue, Y. (2014). "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*, CODASPY '14, ACM, New York, USA, 49–60.

Li, X. and Xue, Y. (2011). "Block: A black-box approach for detection of state violation attacks towards web applications." In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, ACM, New York, USA, 247–256.

Li, X. and Xue, Y. (2013). "Logicscope: Automatic discovery of logic vulnerabilities within web applications." In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIA CCS '13, ACM, New York, USA, 481–486.

Li, X. and Xue, Y. (2014). "A survey on server-side approaches to securing web applications." *ACM Computing Surveys*, 46(4), 54:1–54:29.

Li, X., Yan, W. and Xue, Y. (2012). "Sentinel: Securing database from logic flaws in web applications." In *Proceedings of the Second ACM Conference on Data and Application Security and Privacy*, CODASPY '12, ACM, New York, USA, 25–36.

Liu, A., Yuan, Y., Wijesekera, D. and Stavrou, A. (2009). "SQLProb: A proxy-based architecture towards preventing SQL injection attacks." In *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, ACM, New York, USA, 2054–2061.

Liu, S. and Cheng, B. (2009). "Cyberattacks: Why, what, who, and how." *IT Professional Magazine*, 11(3), 14–21.

Liu, Z. H. and Murthy, R. (2009). "A decade of XML data management: An industrial experience report from oracle." In *IEEE 25th International Conference on Data Engineering, 2009. ICDE '09.*, Shanghai, China, 1351–1362.

MarkLogic. "MarkLogic: Enterprise NoSQL database." `http://www.marklogic.com/` (Apr. 5, 2016).

Medeiros, I., Beatriz, M., Neves, N. and Correia, M. (2016a). "Hacking the DBMS to prevent injection attacks." In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, CODASPY '16, ACM, New York, USA, 295–306.

Medeiros, I., Neves, N. and Correia, M. (2016b). "DEKANT: A static analysis tool that learns to detect web application vulnerabilities." In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, ACM, New York, USA, 1–11.

Menahem, E., Schclar, A., Rokach, L. and Elovici, Y. (2012). "Securing your transactions: Detecting anomalous patterns in XML documents." *arXiv preprint arXiv:1209.1797*.

Meunier, P. (2008). "Classes of vulnerabilities and attacks." *Wiley Handbook of Science and Technology for Homeland Security*.

Microsoft (2015). “Security development lifecycle.” `https://www.microsoft.com/en-us/sdl/process/training.aspx` (Aug. 17, 2015).

Mitropoulos, D., Karakoidas, V., Louridas, P. and Spinellis, D. (2011). “Countering code injection attacks: a unified approach.” *Information Management & Computer Security*, 19(3), 177–194.

Mitropoulos, D., Karakoidas, V. and Spinellis, D. (2009). “Fortifying applications against XPath injection attacks.” In *MCIS 2009: 4th Mediterranean Conference on Information Systems*, Athens, Greece, 1169–1179.

Monshizadeh, M., Naldurg, P. and Venkatakrishnan, V. (2016). “Patching logic vulnerabilities for web applications using LogicPatcher.” In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, CODASPY ’16, ACM, New York, USA, 73–84.

Monshizadeh, M., Naldurg, P. and Venkatakrishnan, V. N. (2014). “Mace: Detecting privilege escalation vulnerabilities in web applications.” In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS ’14, ACM, New York, USA, 690–701.

Morgenstern, J. and Licata, D. R. (2010). “Security-typed programming within dependently typed programming.” In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP ’10, ACM, New York, USA, 169–180.

Mouelhi, T., Le Traon, Y., Abgrall, E., Baudry, B. and Gombault, S. (2011). “Tailored shielding and bypass testing of web applications.” In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation (ICST)*, Berlin, Germany, 210–219.

Muncaster, P. (2014). “Ebay under fire after cross site scripting attack.” `https://www.infosecurity-magazine.com/news/ebay-under-fire-after-cross-site/` (July 26, 2017).

Murdock, J. (2016). “UN tourism website breached and defaced by ’TeamPoison’ hacking collective.” http://www.ibtimes.co.uk/un-tourism-website-breached-defaced-by-teampoison-hacking-collective-1545903 (July 26, 2017).

Muthukumaran, D., O’Keeffe, D., Priebe, C., Eyers, D., Shand, B. and Pietzuch, P. (2015). “FlowWatcher: defending against data disclosure vulnerabilities in web applications.” In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS ’15, ACM, New York, USA, 603–615.

OWASP (2015). “Testing for XML injection.” https://www.owasp.org/index.php/Testing_for_XML_Injection_(OTG-INPVAL-009) (June 6, 2015).

OWASP (2016). “Owasp testing guide version 4.” https://www.owasp.org/images/5/52/OWASP_Testing_Guide_v4.pdf (Sep. 5, 2017).

OWASP (2017a). “SQL injection prevention cheat sheet.” https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet (Sep. 5, 2017a).

OWASP (2017b). “XSS (cross site scripting) prevention cheat sheet.” https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet (Sep. 5, 2017b).

Paganini, P. (2015). “Anonymous hacker breached wto database and leaked data of internal staff.” http://securityaffairs.co/wordpress/36528/hacking/anonymous-breached-wto-db.html (July 26, 2017).

Palsetia, N., Deepa, G., Khan, F. A., Thilagam, P. S. and Pais, A. R. (2016). “Securing native XML database-driven web applications from XQuery injection vulnerabilities.” *Journal of Systems and Software*, 122, 93 – 109.

Parno, B., McCune, J., Wendlandt, D., Andersen, D. and Perrig, A. (2009). “Clamp: Practical prevention of large-scale data leaks.” In *2009 30th IEEE Symposium on Security and Privacy*, Oakland, USA, 154–169.

Pavlovic-Lazetic, G. (2007). "Native XML databases vs. relational databases in dealing with XML documents." *Kragujevac Journal of Mathematics*, 30, 181–199.

Payet, P., Doupé, A., Kruegel, C. and Vigna, G. (2013). "EARs in the wild: Large-scale analysis of execution after redirect vulnerabilities." In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, ACM, New York, USA, 1792–1799.

Pellegrino, G. and Balzarotti, D. (2014). "Toward black-box detection of logic flaws in web applications." In *Proceedings of 21st Network and Distributed System Security Symposium*, NDSS'14, San Diego, USA.

Perkins, J., Eikenberry, J., Coglio, A., Willenson, D., Sidiroglou-Douskos, S. and Rinard, M. (2016). "Autorand: Automatic keyword randomization to prevent injection attacks." In *Detection of Intrusions and Malware, and Vulnerability Assessment: 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings*, Springer International Publishing, Cham, Switzerland, 37–57.

Rosa, T. M., Santin, A. O. and Malucelli, A. (2013). "Mitigating XML injection 0-day attacks through strategy-based detection systems." *IEEE Security & Privacy*, 11(4), 46–53.

SANS (2011). "CWE/SANS top 25 most dangerous software errors." `http://www.sans.org/top25-software-errors/` (Nov. 8, 2016).

Saxena, P., Molnar, D. and Livshits, B. (2011). "Scriptgard: Automatic context-sensitive sanitization for large-scale legacy web applications." In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, ACM, New York, USA, 601–614.

Scholte, T., Balzarotti, D. and Kirda, E. (2012a). "Have things changed now? An empirical study on input validation vulnerabilities in web applications." *Computers & Security*, 31(3), 344–356.

Scholte, T., Robertson, W., Balzarotti, D. and Kirda, E. (2012b). "Preventing input validation vulnerabilities in web applications through automated type analysis." In *2012*

IEEE 36th Annual Computer Software and Applications Conference (COMPSAC), Izmir, Turkey, 233–243.

Shahriar, H. and Devendran, V. K. (2014). “Classification of clickjacking attacks and detection techniques.” *Information Security Journal: A Global Perspective*, 23(4-6), 137–147.

Shahriar, H., Devendran, V. K. and Haddad, H. (2013). “Proclick: A framework for testing clickjacking attacks in web applications.” In *Proceedings of the 6th International Conference on Security of Information and Networks*, SIN ’13, ACM, New York, USA, 144–151.

Shahriar, H. and Haddad, H. (2014). “Risk assessment of code injection vulnerabilities using fuzzy logic-based system.” In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC ’14, ACM, New York, USA, 1164–1170.

Shahriar, H., North, S., Chen, W.-C. and Mawangi, E. (2014a). “Information theoretic XSS attack detection in web applications.” *International Journal of Secure Software Engineering*, 5(3), 1–15.

Shahriar, H., North, S. M., Lee, Y. and Hu, R. (2014b). “Server-side code injection attack detection based on Kullback-Leibler distance.” *International Journal of Internet Technology and Secured Transactions*, 5(3), 240–261.

Shahriar, H. and Zulkernine, M. (2009). “Mutec: Mutation-based testing of cross site scripting.” In *ICSE Workshop on Software Engineering for Secure Systems, SESS ’09*, Vancouver, Canada, 47–53.

Shahriar, H. and Zulkernine, M. (2011a). “S2XS2: A server side approach to automatically detect XSS attacks.” In *2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing (DASC)*, Sydney, Australia, 7–14.

Shahriar, H. and Zulkernine, M. (2011b). “Taxonomy and classification of automatic monitoring of program security vulnerability exploitations.” *Journal of Systems and Software*, 84(2), 250–269.

Shahriar, H. and Zulkernine, M. (2012a). “Information-theoretic detection of SQL injection attacks.” In *2012 IEEE 14th International Symposium on High-Assurance Systems Engineering (HASE)*, Omaha, USA, 40–47.

Shahriar, H. and Zulkernine, M. (2012b). “Mitigating program security vulnerabilities: Approaches and challenges.” *ACM Computing Surveys*, 44(3), 11:1–11:46.

Shar, L. K. and Tan, H. B. K. (2012). “Automated removal of cross site scripting vulnerabilities in web applications.” *Information and Software Technology*, 54(5), 467–478.

Shar, L. K. and Tan, H. B. K. (2013). “Predicting SQL injection and cross site scripting vulnerabilities through mining input sanitization patterns.” *Information and Software Technology*, 55(10), 1767–1780.

Shura, B. (2010). “Web application security scanner list.” `http://projects.webappsec.org/w/page/13246988/Web%20Application%20Security%20Scanner%20List` (June 6, 2015).

Siegel, E. and Retter, A. (2014). *eXist*, O’Reilly Media, USA.

Sima, C. (2003). “Security at the next level: Are your web applications vulnerable?.” *SPI Labs*.

Skrupsky, N., Bisht, P., Hinrichs, T., Venkatakrishnan, V. N. and Zuck, L. (2013). “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*, CODASPY ’13, ACM, New York, USA, 129–140.

Son, S., McKinley, K. S. and Shmatikov, V. (2011). “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*, OOPSLA ’11, ACM, New York, USA, 1069–1084.

Son, S., McKinley, K. S. and Shmatikov, V. (2013). "Fix me up: Repairing access-control bugs in web applications." In *Proceedings of 20th Annual Network and Distributed System Security Symposium*, NDSS'13, San Diego, USA.

Son, S. and Shmatikov, V. (2011). "SAFERPHP: Finding semantic vulnerabilities in PHP applications." In *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security*, PLAS '11, ACM, New York, USA, 8:1–8:13.

Staken, K. (2001). "Introduction to native XML databases." `http://www.xml.com/pub/a/2001/10/31/nativexmldb.html` (Apr. 5, 2016).

Stock, B., Lekies, S., Mueller, T., Spiegel, P. and Johns, M. (2014). "Precise client-side protection against DOM-based cross-site scripting." In *Proceedings of the 23rd USENIX security symposium*, USENIX Association, San Diego, USA, 655–670.

Stuttard, D. and Pinto, M. (2011). *The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws*, John Wiley & Sons, USA.

Su, Z. and Wassermann, G. (2006). "The essence of command injection attacks in web applications." In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, ACM, New York, USA, 372–382.

Sun, F., Xu, L. and Su, Z. (2011). "Static detection of access control vulnerabilities in web applications." In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, USENIX Association, Berkeley, USA, 11–11.

Sun, F., Xu, L. and Su, Z. (2014). "Detecting logic vulnerabilities in e-commerce applications." In *Proceedings of 21st Network and Distributed System Security Symposium*, NDSS'14, San Diego, USA.

Swamy, N., Chen, J. and Chugh, R. (2010). "Enforcing stateful authorization and information flow policies in fine." In *Programming Languages and Systems*, volume 6012 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 529–549.

Swamy, N., Corcoran, B. and Hicks, M. (2008). "Fable: A language for enforcing user-defined security policies." In *2008 IEEE Symposium on Security and Privacy*, Oakland, USA, 369–383.

Sweet, L. (2017). "Illinois' chapter in the Russian hacking saga." `http://chicago.suntimes.com/news/illinois-chapter-in-the-russian-hacking-saga/` (July 26, 2017).

Symantec (April 2014). "Symantec Internet Security Threat Report." Technical report, Symantec Inc.

Symantec (April 2016). "Symantec Internet Security Threat Report." Technical report, Symantec Inc.

Ter Louw, M. and Venkatakrishnan, V. (2009). "Blueprint: Robust prevention of cross-site scripting attacks for existing browsers." In *2009 30th IEEE Symposium on Security and Privacy*, Oakland, USA, 331–346.

Thomas, S. and Williams, L. (2007). "Using automated fix generation to secure SQL statements." In *Third International Workshop on Software Engineering for Secure Systems. SESS '07: ICSE Workshops 2007*, Minneapolis, USA, 9–9.

Top10 (2013). "OWASP Top Ten 2013." Technical report, OWASP Foundation.

Truelove, J. and Svoboda, D. (2011). "IDS09-J. Prevent XPath injection." `https://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=61407250` (Apr. 5, 2016).

Trustwave. "Trustwave app scanner." `https://www.trustwave.com/Products/Application-Security/App-Scanner-Family/` (June 6, 2015).

Trustwave (2011). "2011 Trustwave global security report." Technical report, Trustwave.

Trustwave (2014). "2014 Trustwave global security report." Technical report, Trustwave.

Trustwave (2016). “2016 Trustwave global security report.” Technical report, Trustwave.

Tsipenyuk, K., Chess, B. and McGraw, G. (2005). “Seven pernicious kingdoms: a taxonomy of software security errors.” *IEEE Security & Privacy*, 3(6), 81–84.

Valeur, F., Mutz, D. and Vigna, G. (2005). “A learning-based approach to the detection of SQL attacks.” In *Proceedings of the Second International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA’05, Springer-Verlag, Berlin, Heidelberg, 123–140.

Van Acker, S., Nikiforakis, N., Desmet, L., Joosen, W. and Piessens, F. (2012). “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*, ASIACCS ’12, ACM, New York, USA, 12–13.

van der Loo, F. (2011). “Comparison of penetration testing tools for web applications.” Master’s thesis, Radboud University Nijmegen. http://www.ru.nl/publish/pages/769526/frank_van_der_loo_scriptie.pdf.

Van Gundy, M. and Chen, H. (2009). “Noncespaces: Using randomization to enforce information flow tracking and thwart cross-site scripting attacks.” In *Proceedings of 16th Network and Distributed System Security Symposium*, NDSS’09, San Diego, USA.

Van Gundy, M. and Chen, H. (2012). “Noncespaces: Using randomization to defeat cross-site scripting attacks.” *Computers & Security*, 31(4), 612–628.

Verizon (2014). “Data breach investigations report.” Technical report, Verizon.

Vikram, K., Prateek, A. and Livshits, B. (2009). “Ripley: Automatically securing web 2.0 applications through replicated execution.” In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS ’09, ACM, New York, USA, 173–186.

Vogt, P., Nentwich, F., Jovanovic, N., Kirda, E., Kruegel, C. and Vigna, G. (2007). "Cross site scripting prevention with dynamic data tainting and static analysis." In *Proceedings of the 14th Network and Distributed System Security Symposium*, NDSS'07, San Diego, USA.

W3C (2015). "XML security." `https://www.w3.org/standards/xml/security` (Apr. 5, 2016).

Wang, R., Chen, S. and Wang, X. (2012). "Signing me onto your accounts through facebook and google: A traffic-guided security study of commercially deployed single-sign-on web services." In *2012 IEEE Symposium on Security and Privacy*, San Francisco, USA, 365–379.

Wang, R., Chen, S., Wang, X. and Qadeer, S. (2011). "How to shop for free online – security analysis of cashier-as-a-service based web stores." In *2011 IEEE Symposium on Security and Privacy*, Oakland, USA, 465–480.

Ward, J. (2007). "What is a rich internet application?." `http://www.jamesward.com/2007/10/17/what-is-a-rich-internet-application/`).

Wassermann, G. and Su, Z. (2007). "Sound and precise analysis of web applications for injection vulnerabilities." In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, ACM, New York, USA, 32–41.

Wassermann, G. and Su, Z. (2008). "Static detection of cross-site scripting vulnerabilities." In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, ACM, New York, USA, 171–180.

WebCruiser (2011). "Webcruiser-web vulnerability scanner." `http://www.ehacking.net/2011/07/webcruiser-web-vulnerability-scanner.html` (Apr. 5, 2016).

Wedman, S., Tetmeyer, A. and Saiedian, H. (2013). "An analytical study of web application session management mechanisms and HTTP session hijacking attacks." *Information Security Journal: A Global Perspective*, 22(2), 55–67.

Weinberger, J., Saxena, P., Akhawe, D., Finifter, M., Shin, R. and Song, D. (2011). “A systematic analysis of XSS sanitization in web application frameworks.” In *Computer Security ESORICS 2011*, volume 6879 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 150–171.

Wen, S., Xue, Y., Xu, J., Yang, H., Li, X., Song, W. and Si, G. (2016). “Toward exploiting access control vulnerabilities within MongoDB backend web applications.” In *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, Atlanta, USA, 143–153.

Whitehat (2014). “2014 Website Security Statistics Report.” Technical report, WhiteHat Security Inc.

Win, B. D. (2014). “Secure development lifecycles (SDLC).” `http://secappdev.org/handouts/2014/Bart%20De%20Win/SDLC%20v1.0.pdf` (Aug. 17, 2015).

Wurzinger, P., Platzer, C., Ludl, C., Kirda, E. and Kruegel, C. (2009). “SWAP: Mitigating XSS attacks using a reverse proxy.” In *ICSE Workshop on Software Engineering for Secure Systems, SESS ’09*, Vancouver, Canada, 33–39.

Xie, Y. and Aiken, A. (2006). “Static detection of security vulnerabilities in scripting languages.” In *Proceedings of the 15th USENIX Security Symposium - Volume 15*, USENIX-SS’06, USENIX Association, Berkeley, USA.

Xing, L., Chen, Y., Wang, X. and Chen, S. (2013). “Integuard: Toward automatic protection of third-party web service integrations.” In *Proceedings of 20th Annual Network and Distributed System Security Symposium*, NDSS’13, San Diego, USA.

XPath-Injection (2015). “XPath injection.” `https://www.owasp.org/index.php/XPATH_Injection` (Apr. 5, 2016).

XXE (2016). “XML external entity (XXE) processing.” `https://www.owasp.org/index.php/XML_External_Entity_(XXE)_Processing` (Apr. 5, 2016).

Yip, A., Wang, X., Zeldovich, N. and Kaashoek, M. F. (2009). "Improving application security with data flow assertions." In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, ACM, New York, USA, 291–304.

ZAP (2016). "OWASP zed attack proxy project." `https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project` (Apr. 5, 2016).

Zetter, K. (2011). "Citi credit card hack bigger than originally disclosed." `https://www.wired.com/2011/06/citibank-hacked/` (July 26, 2017).

Zhu, J., Chu, B., Lipford, H. and Thomas, T. (2015). "Mitigating access control vulnerabilities through interactive static analysis." In *Proceedings of the 20th ACM Symposium on Access Control Models and Technologies*, SACMAT '15, ACM, New York, USA, 199–209.

PUBLICATIONS

1. Deepa, G. and Thilagam, P. S. (2016). "Securing web applications from injection and logic vulnerabilities: Approaches and challenges." *Information and Software Technology*, Elsevier, 74, 160–180.
   URL: `http://dx.doi.org/10.1016/j.infsof.2016.02.00`

2. Palsetia, N., Deepa, G., Khan, F. A., Thilagam, P. S. and Pais, A. R. (2016). "Securing native XML database-driven web applications from XQuery injection vulnerabilities." *Journal of Systems and Software*, Elsevier, 122, 93–109.
   URL: `http://doi.org/10.1016/j.jss.2016.08.094`

3. Deepa, G., Thilagam, P. S., Khan, F. A., Praseed, A., Pais, A. R. and Palsetia, N. (2018). "Black-box detection of XQuery injection and parameter tampering vulnerabilities in web applications." *International Journal of Information Security*, Springer, 17(1), pages 105–120.
   URL: `http://dx.doi.org/10.1007/s10207-016-0359-4`

4. Deepa, G., Thilagam, P. S., Praseed, A. and Pais, A. R. (2018). "DetLogic: A black-box approach for detecting logic vulnerabilities in web applications." *Journal of Network and Computer Applications*, Elsevier, 109, 89–109.
   URL: `https://doi.org/10.1016/j.jnca.2018.01.008`

Bio-Data

Personal Data

Name:	G Deepa
Date of Birth:	21 October 1984
Sex:	Female
Marital Status:	Married
Father's Name:	K Ganesan
Mother's Name:	G Saratha
Address:	AP-3 NITK Quarters, NITK Surathkal, Mangaluru, Karnataka, India - 575 025.
Phone:	+91 8951261510
email:	gdeepabalu@gmail.com

Education

2012 Master of Engineering, **Anna University**, Chennai, Tamil Nadu
Specialization: Computer Science and Engineering
CGPA: 8.86

2006 Bachelor of Engineering, **Anna University**, Chennai, Tamil Nadu
Specialization: Electrical and Electronics Engineering
CGPA: 9.3

Professional Experience

June 2013 - Oct 2013	Assitant Professor at Shivani Engineering College, Trichy
Aug 2006 - Aug 2012	IT Analyst at Tata Consultancy Services, Chennai

Awards and Honors

2012	University Rank Holder (7th Rank in M.E.)
2006	University Gold Medalist (B.E.)
2002	School topper in Higher Secondary Examination
	Scored Centum in Mathematics and Physics in Higher Secondary Examination

Areas of Interest

Web Application Security, Software Testing, Data Mining