بسم الله الرحمن الرحيم
١٤١٨

AGENT BASED SECURITY TESTING OF WEB BASED APPLICATIONS

Muhammad Imran

A thesis submitted for the requirements of the degree of Master of Science in Computer Science

Supervised By

Prof. Dr. Fathy Elbouraey Eassa

DEPARTMENT OF COMPUTER SCIENCE
FACULTY OF COMPUTING AND INFORMATION TECHNOLOGY
KING ABDULAZIZ UNIVERSITY
JEDDAH – SAUDI ARABIA
Jumada II, 1435 H – April 2014 G

AGENT BASED SECURITY TESTING OF WEB BASED APPLICATIONS

By

Muhammad Imran

This thesis has been approved and accepted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

EXAMINATION COMMITTEE APPROVAL

	Name	Rank	Field	Signature
Internal Examiner				
External Examiner				
Supervisor				

KING ABDULAZIZ UNIVERSITY
Jumada II, 1435 H – April 2014 G

DEDICATION

I would like to dedicate this work to my beloved parents (God's mercy be upon them), my brother and my sisters for their deep love, high encouragement and priceless prayers and support for my success throughout the study.

ACKNOWLEDGMENT

First of all, thanks to Allah Almighty who kept his help with me granted me the strength to accomplish this work. In completing this thesis, I have been fortunate to receive help, support and encouragement from many people. I would like to acknowledge them for their help.

I would like to thank Prof. Dr. Fathy Elbouraey Eassa, my thesis advisor, for guiding me through each step of the process with practical knowledge and support. Thank you for your advice, guidance and assistance.

I would also like to extend my gratitude towards my colleagues and friends, Muhammad Aqib, Fahad Muhammad Bahazaq and Omer Farooqi who were the most helpful during this difficult task. Also my roommates, Ali Ayaz Hussain Kazmi, Safi Asim bin Asif, Mazhar-ullah Lodhi, and Haris Ahmed Khan were great supporters throughout my research.

A very special thanks goes to my loving and supporting family and most respectfully to my father and mother. Without their continuous motivation and prayers I would not have been able to complete this task.

AGENT BASED SECURITY TESTING OF WEB BASED APPLICATIONS

Muhammad Imran

ABSTRACT

With the advancement in technologies more and more services and operations are being deployed in the World Wide Web. This is the reason of increased demand, development and use of web applications so as the number of web attacks and web application security vulnerabilities have been raised. Security plays an important role in software and the field of web application security has been matured in the previous years, whereas the vulnerabilities keep on evolving and attack methods keep on getting refined getting benefit of bad programming practices. This creates a gap for more optimized security testing technique and tool to detect and prevent the security attacks from happening.

Among so many other classes and categories, a class of security vulnerabilities known as input validation vulnerabilities exists which are caused because of improper user input validation where the attack is initiated by the malicious input ending up executed by a sensitive vulnerable function.

In this thesis, we introduce an optimized agent based integrated static and dynamic analysis technique and tool for detecting and preventing such security vulnerabilities in web applications written in Java. We start with the static taint analysis for tracking the propagation of user input in the program which helps to detect the vulnerabilities in the source code. In the next phase, the dynamic analysis is used for prevention of attacks and reduction of false positives generated by the static analysis. Our technique is extendable to the vulnerabilities in the similar class and source codes written in other object oriented languages. Finally, we have also developed our proposed technique as an agent based security testing tool called Java Web Application Security Tester.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND TERMINOLOGY

OWASP	Open Web Application Security Project
XSS	Cross-Site Scripting
CSRF	Cross-Site Request Forgery
RFI	Remote File Inclusion
SQLI	SQL Injection
LDAPI	Lightweight Directory Access Protocol Injection
JavaCC	Java Compiler Compiler
JIE	Java Instrumentation Engine
EBNF	Extended Backus-Naur Form
AST	Abstract Syntax Tree
CFG	Control Flow Graph
JWAST	Java Web Application Security Tester

Chapter 1

Introduction

Web applications are the client/server applications in which web browser is used as a client and interactive service is performed by connecting with the servers over the internet [1]. Web applications have been evolved in their need and architecture since the early days to of the internet. From banking to e-shopping and government healthcare systems, huge amount of operations are done using web based applications [2]. The use of web based applications has been growing as the speed and availability of internet is increasing and more and more services are being deployed in the World Wide Web and according to the reports and surveys, in 2013 hardly any organization would not be making use of web based applications [3]. Web applications are naturally completely exposed and are easy to access through the browsers, which is why they are more prone to vulnerabilities and cyber-attacks and on the networks.

The diverse use of web applications has literally opened a can of bugs. This is due to different number of reasons and one of them is the underlying communication protocol i.e. HTTP, typically used by the web applications. The Hyper Text Transmission Protocol (HTTP) is a stateless protocol and is itself vulnerable. Whereas, this protocol is used to access all the data required from a web application. Also the web server, where a web application is deployed and accessed by its users, also exhibits a vulnerable nature where it is present at a public web hosting site which do not enforce any type of quality assurance on the web applications that are hosted there. These factors increases the chances for a web application to be attacked. The research shows that the application level attacks are a focus for hackers as most of the attacks come from the application level [2]. Whereas, the attack types, attack impacts and vulnerabilities are rapidly changing by the passage of time as shown in Figure 1.1, Figure 1.2 and Figure 1.3. Upon comparing these figures we can easily find out the increasing number, type and impact of vulnerabilities over 3 years where the size of circle estimates relative impact of incident in terms of cost to business.

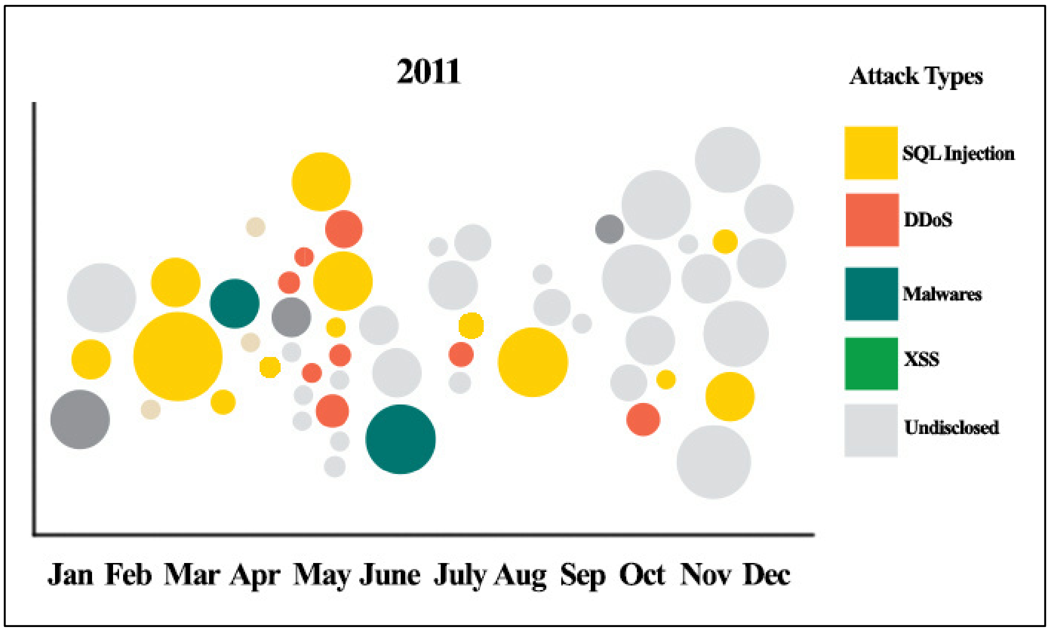

Figure 1.1: Security Incidents by Attack Type, Time and Impact in 2011 [4]

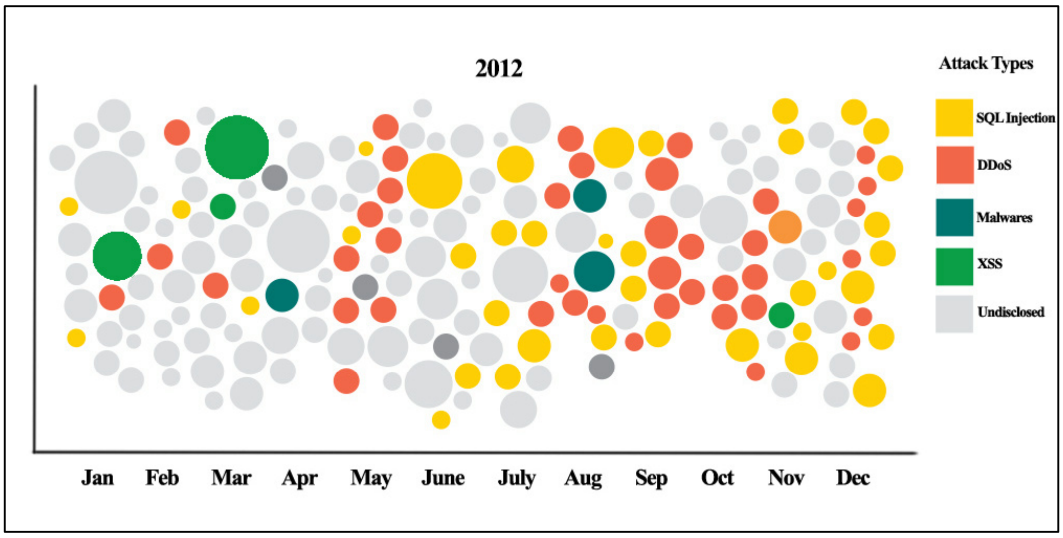

Figure 1.2: Security Incidents by Attack Type, Time and Impact in 2012 [4]

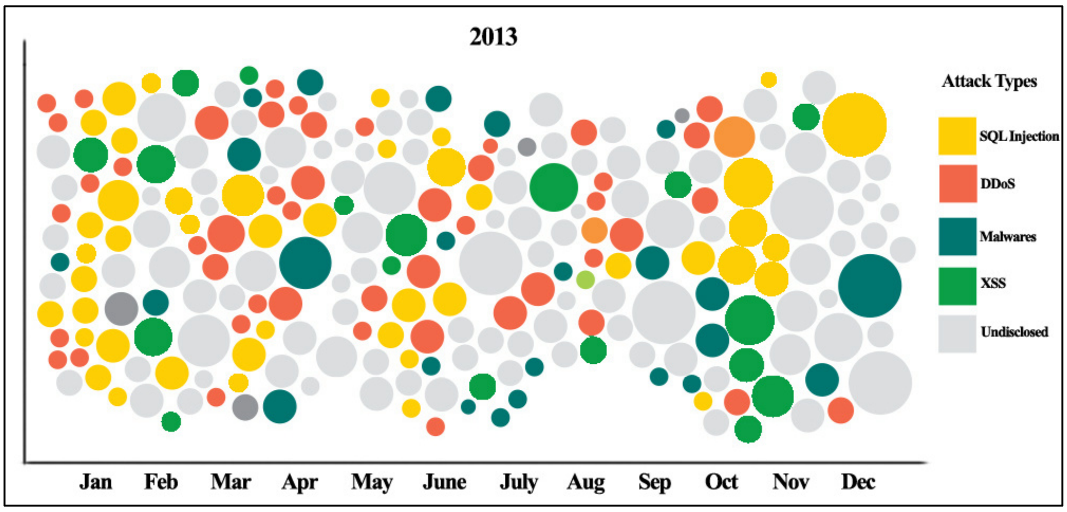

Figure 1.3: Security Incidents by Attack Type, Time and Impact in 2013 [4]

1.1 Motivation

In the research field of web applications security different authors have worked on several mechanisms to make the web applications more secure. However the field of security testing is not yet matured and there is need of testing methodologies, techniques and tools [5]. The number of vulnerabilities that are detected every year are on the increasing trend where each year the disclosed number of vulnerabilities are more than the preceding year. Figure 1.1 shows the growth rate of vulnerabilities disclosed where the total number of vulnerabilities disclosed have been increasing since the beginning. Not only the number increases but also type of vulnerabilities are increasing as the types of attacks are becoming more sophisticated by the passage of time. Web applications being more complex than a traditional application needs different approach for testing and among other non-functional requirements like performance, scalability and usability and so on, security is also very important and critical feature which needs to be tested [6].

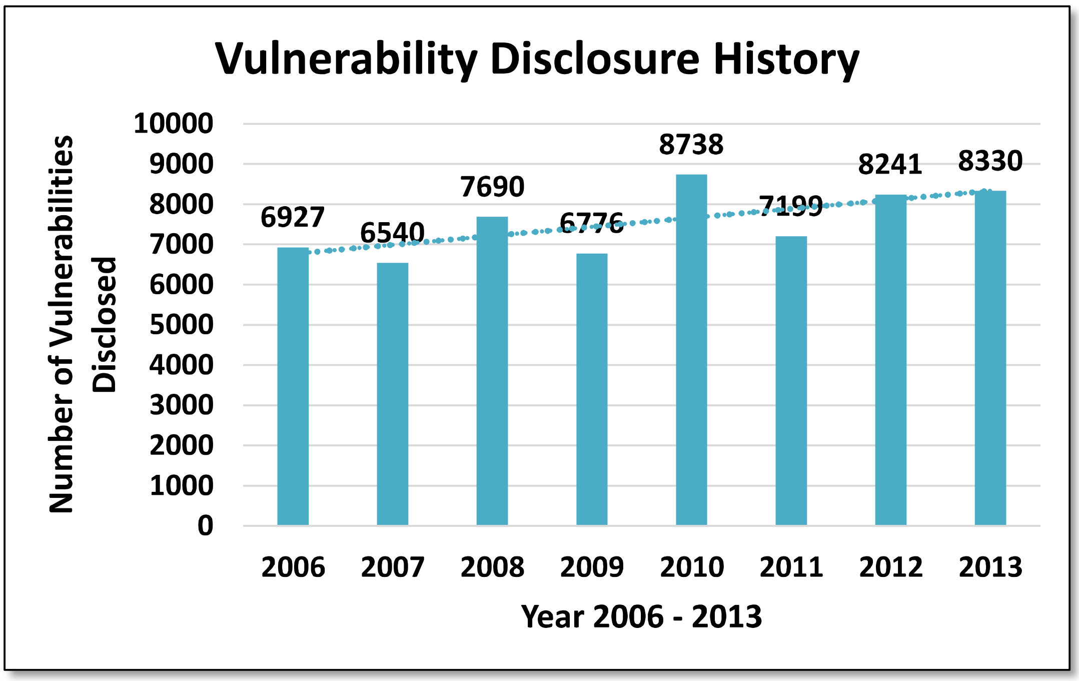

Figure 1.4: Vulnerability disclosures growth by Year (2006 - 2013) [4]

Increase in the web application vulnerabilities and attacks are causing damage to the web application layer and affecting the performance. On the internet, the network is not enough to protect the web application layer and so we need to design the web application security using new paradigms. In this thesis, we will be developing a security solution using a non-conventional approach i.e. Mobile Agents. This project will allow us to provide an integrated mechanism of testing the web applications using the mobile agents, static analysis techniques and dynamic analysis techniques and the use of mobile agents will power up our tool by providing their features like asynchronous nature, dynamic instantiation, portability and platform independence, enabling our tool to achieve several performance goals like reducing the network load, performing concurrent testing and so on.

1.2 Scope of the Thesis

This thesis is focused on building a technique and developing a tool to detect and prevent the security vulnerabilities in the web based applications. Therefore, security vulnerabilities to be tackled by this project, should be identified and prioritized. Different vulnerability classes, categories and rankings already exist, which are prepared and maintained by different projects and organizations like Open Web Application Security Project (OWASP). Based on the target vulnerabilities set, the next part should be concerned with the identification of suitable static analysis technique followed by an appropriate dynamic analysis technique, using the vulnerabilities knowledge gained in the first part. The identified techniques should be optimized separately and should be further integrated forming an integrated static and dynamic analysis technique. For making the solution more adaptable and powerful, an agent based mechanism should be devised which could incorporate the already developed integrated analysis technique. Afterwards the integrated analysis technique should be implemented as an agent based tool to perform the evaluation of the proposed solution. Later on, the developed tool needs to be tested for the validation purposes making sure the target vulnerability classes are being detected and prevented.

1.3 The Objectives

In this thesis the goal will be to develop an optimized technique based on integration of static analysis and dynamic analysis for the detection and prevention of security vulnerabilities in the web based applications.

For the achievement of our goal, following will be the objectives of our research:

1. Identifying and deciding the current Top 10 security vulnerabilities.
2. Developing mechanism for static analysis and dynamic analysis for the identified vulnerabilities.

3. Agent-based integration mechanism for the static analysis and dynamic analysis approaches to detect and prevent the vulnerabilities.
4. Building an agent based security testing tool to implement and evaluate our technique.
5. Testing the developed security tool and comparing it with others.

1.4 The Methodology

For the achievement of our goal and the objectives mentioned in the previous section, we followed the methodology for our research as described below:

1. Doing the literature review for the related work and building the background about Web Application Security.
2. Identifying the different types of web application security vulnerabilities (Top Ten) and their features.
3. Building a mechanism for web application's static analysis to detect some vulnerabilities in the source code.
4. Building a mechanism for web application's dynamic analysis for detecting and preventing runtime security attacks.
5. Building and implementing an integration tool based on software agents to integrate the two (static and dynamic) analysis mechanisms.
6. Evaluating our solution and tool for the web application security.
7. Comparing our tool with the already existing other tools used for the same purpose.

1.5 Thesis Outline

The rest of this thesis is organized into number of chapters where each chapter tries to cover a specific part of the problem under consideration by dividing it into small sections. Chapter 2 focuses on the required background knowledge that would be helpful in solving the security problem. Specially, the web applications vulnerabilities are the main focus of this chapter where the preliminary study about the web application vulnerabilities has been presented and this study has been further applied to the later chapters. We begin Chapter 3 by discussing the related works that have been done on the same subject as of this thesis. A brief and comprehensive review on the existing tools meant for security by static or dynamic or both analyses has been added in this chapter. Chapter 4 describes our proposed solution with its details including the detailed architecture of our security tool. The implementation details related to the security tool presented in this thesis are provided in Chapter 5. The test cases that were run for testing the tool are also added in this chapter. The evaluation and comparisons of our tool with other already existing security tools are presented in Chapter 6. Finally, Chapter 7 concludes this thesis and discusses some enhancements and extensions that are kept for future additions as a future work.

Chapter 2

Background

In this chapter some topics that are important to be understood are described in detail. First the context of web application security has been explained according to our current study. Second some security testing techniques that are in practice are described briefly and then an introduction to the static analysis and dynamic analysis techniques is provided as a foundation for building our own integrated technique in chapter 4. Later in the chapter a comprehensive study on the web application vulnerabilities is presented for the understanding of vulnerabilities that are targeted by our security testing tool.

2.1 Web Application Security Context

The web application security has become a concern not only for individual people and private organizations but also the government institutions have started researching about it and discussing about major attacks. One of the reports by InfoSec [7] not only stated the counter measures but also recommended running regular security tests. Different works and projects target the web application security issue. A recent vulnerability trends report by CENZIC presents the current vulnerabilities and risk landscape. Out of all the applications tested by them, 99% had one or more serious security vulnerabilities but they were also detectable, preventable and fixable. Among other common vulnerabilities, Cross Site Scripting (XSS) topped the list of vulnerabilities detected and not only it topped the list, its level was also on the rise from 2011 to 2012 which was from 17% to 26% [3].

Also the importance of the web application security testing can be understood as in the recent years, a large variety of tools has been proposed to be used for the security testing of web applications among which both the commercial and free use tools are available. Around 50 tools are list at the Software QA and Testing Resource Center [8] for web application security testing. Like many others, CENZIC [3] also recommends security testing and frequent scanning of web applications to anticipate the future vulnerabilities in the application.

2.2 Web Application Security Testing Techniques

The security problem of detecting the vulnerabilities can be addressed in different possible ways. Depending on the security requirements, critical level and other available resources, the testing approach can vary like Agile Security Testing [9], Penetration Testing [10] and other approaches. The two commonly used broad categories for security testing are static and dynamic analysis and each approach for security testing can be related to these categories.

Several testing techniques exist that cannot be considered in the boundary of static analysis or dynamic analysis techniques. Some of these are explained in the following sections.

2.2.1 Intrusion Detection

Intrusion detection systems can be used for the detection of attacks in web based applications and the servers they are hosted on. Different techniques for detecting the irregularities can be applied some of which like multi modal approach [11] and anomaly detections of web based attacks [12] are already present in the literature. For the intrusion detection, the client side queries are analyzed if they are referring the server side program for example the use of access patterns of server side programs which might be learned by getting training on some past data.

2.2.2 Protecting the Client Side

One of the techniques for securing the web application is to add protection to the web application client. These techniques for web applications usually target the clients which are the web browsers in typical scenarios. A proxy is placed between the browser and those client requests and all the authentication credentials are filtered by the application server which are identified as potential cross site request forgery (CSRF) attacks. An example of such proxy based system is RequestRodeo [13]. Also another browser based technology called Noxes [14] also provides protection against the information leakage which is actually a side effect of cross site scripting (XSS) attacks. Similarly, Paros [15] is a Java based HTTP/HTTPS proxy for assessing web application vulnerability. It supports editing/viewing HTTP messages on-the-fly. Other features include spiders, client certificate, proxy-chaining, and intelligent scanning for XSS and SQL injections.

2.2.3 System Design for Better Application Security

One of the techniques that is used for securing the web and other software applications include embedding the security into the design and architecture of the system. Several works have followed this approach to design their frameworks like Tahoma [16]. One example of a security measure that can be taken by using this framework is that the publisher of web application can restrict a set of URLs for a particular application and in this way a level of isolation will be provided between the web application and the operating system. Another very similar approach is used in an architecture known as Terra [17].

2.3 Static Analysis

Static analysis which might also be known as static program analysis or static code analysis is “the analysis of a computer software which is performed without actually executing the software under testing” [18].The program’s text is statically examined in this analysis and a possibility of applying the static analysis on the compiled form of the program also exists but decoding can be a problem in this case. Manual auditing and Code Review might also fall under the category of static analysis but this remains ineffective until and unless this activity is automated making it faster and reliable [19]. The automated static analysis tools and techniques exist which evaluate the programs more frequently. In addition, the static analysis can also be applied early in the software development lifecycle before the completion and testing phase of the program by analyzing the code fragments or modules giving feedback to the software developers and enabling cheaper solutions. Security vulnerabilities are targeted by some of the static analysis tools whereas rest of them are meant for other problems in the programs. Static analysis has its application for multiple purposes including type checking in programs, style and property checking, finding bugs, code beautification (like code indentations), code inspection, Pattern based analysis, Metrics based analysis (like checking the code complexity or total number of lines or methods) and finding security vulnerabilities which is a growing use of this approach in response to the growing number of vulnerable software systems. [20].

With the static analysis approach it cannot be guaranteed that all the vulnerabilities in the program are detected which is due to the fact that in the worst case, static analysis problems are undecidable and they tend to produce false negatives by not reporting the bugs that program has or false positives by reporting the bugs that program does not have [19].

2.4 Dynamic Analysis

Runtime monitoring falls under the category of dynamic analysis and this refers to monitoring the program under test during its execution. Further, different techniques exist for achieving the runtime monitoring. However, the techniques of runtime monitoring are more feasible for preventing the runtime attacks rather than their detection. So, it cannot be used for detecting the location of vulnerabilities but it can help in preventing the vulnerability to be exploited during the program execution.

Similarly, another set of techniques that also falls under the category of dynamic analysis is known as penetration testing [10]. In penetration testing the system is considered a blackbox. A set of input variables, whose values are set as some malicious inputs, is composed manually or automatically and is given to the program under test. And finally the behavior of the program as a result of that input is evaluated. However, this technique depends on the set of input values that are given and setting these input requires a significant security knowledge.

2.5 Web Application Vulnerabilities

Any malicious or other potential occurrence which can damage the software execution, operation and its use can be referred to as a threat. This threat is actually made possible by a vulnerability, where this vulnerability can be a security flaw, defect, or mistake in software that can be directly used by a by the attacker to gain access to a system or network. Further, after the existence of the vulnerability in the software, if that vulnerability is exploited by a malicious user, an attack is said to be attempted.

Several research studies has been carried out which can broadly be categorized into qualitative and quantitative studies. As our research is based on the security testing of web based applications, understating the web application vulnerabilities, prior to proposing a security testing technique, is of great importance. In the following sub-sections we have presented different ways used to understand, categorize and manage the web application vulnerabilities.

We have narrowed down our target number and type of vulnerabilities by studying the available rankings and categorizations in the literature. The vulnerabilities that are targeted by our tool have been explained in detail including their behavior and examples in the later sub-sections below.

2.5.1 Ranking the Most Critical

Based on the several factors like most occurrences, critical level, number of attacks reported in certain period of time and so on, there exist several categorizations made by different organization and projects that have been working on the web application security problem. Some of these categorizations include OWASP Top 10 [21]. It is a project managed under the Open Web Application Security Project (OWASP) [22], an organization which is focused on improving the security of software. The OWASP releases a yearly report which ranks the most critical web application vulnerabilities for that year. OWASP Top 10 categorizes the “Injection (SQL, OS, and LDAP)” as the most vulnerable in the Top 10 security risks. Other similar rankings, statistics and reports are published by Imperva’s Web Application Attack Report, Cenzic Application Vulnerability Trends Report, Context Information Security, and IBM X-Force Threat Intelligence.

Attack report from Imperva [23] also presented the current state of web security vulnerabilities and attach statistics mentioning the worst case attack incident lasted for 80 minutes and it happened on 80% of the days that at least one attack incident occurred. The report focuses on specific attack types among which the SQL Injections are with the highest incident magnitude and Remote File Inclusion (RFI) attacks are the longest duration attacks to happen. Use of security solutions for detection and prevention of security vulnerabilities has been recommended. Other projects and organization are also working in this direction which can be used in targeting the most vulnerable ones. A white paper from Context [24] documents a very detailed statistics report categorizing the vulnerabilities into groups with server misconfiguration having highest average.

2.5.2 Classification of Vulnerabilities

"A vulnerability class is a set of vulnerabilities that share some unifying commonality pattern or concept that isolates a specific feature shared by several different software flaws" [25].

The identified number of security vulnerabilities has been increasing and continue to increase. To better understand and have an abstract overview of these vulnerabilities, classifications in the form of taxonomies are introduced by different researchers. Also, this distinction among the vulnerabilities in the form of classifications help in examining the nature and extent of problem in a better way. The initial classification was made by Landwehr [26]. This taxonomy divided the flaws on the basis of origin of vulnerabilities, time of vulnerability origin, and the location of vulnerability in the program.

Recently, a fair number of woks have tried to classify the vulnerabilities. "The 19 Deadly Sins of Software Security" [27] claims to cover 95% of all the security issues and it mentioned a total of 19 common security defects. As a further addition this work has been extended later on and where five more defects are added to the previously mentioned 19 defects and classified as "24 Deadly Sins of Software Security" [28].

The taxonomy provided by the “Seven Pernicious Kingdoms” [29] also classifies the vulnerabilities based on different criteria which may include the vulnerability behavior, attacking pattern and attack targets that are achievable by exploiting a vulnerability. This taxonomy classifies hierarchy into seven categories, where each category is called as a kingdom. Also, the classification by this taxonomy have been the main source for our study and we have targeted the “Input Validation and Representation” vulnerability class to be tested for security in the web based applications. Other classes that are included in this taxonomy are API Abuse, Security Features, Time and State, Errors, Code Quality, and Encapsulation.

2.5.3 Input Validation Vulnerabilities

In this study, we have targeted one of the classes of web application vulnerabilities known as “Input Validation Vulnerabilities”. Our developed security testing tool Java Web Application Security Tester (JWAST) is cable of detecting such vulnerabilities in the web based applications written in Java. The list of vulnerabilities that our tool can detect are listed in Table 2.1. Also, the detailed explanation of these vulnerabilities is given in this section. “Input validation vulnerabilities” lies among one of the groups which are classified by Seven Pernicious Kingdom taxonomy [29] and OWASP [22]. This category also includes many top vulnerabilities listed by CWE/SANS [30] like buffer overflows, Cross-site scripting, SQL injections, format string, integer overflow, etc.

Vulnerability Type	Malicious Input Example
SQL Injection	' or '1'='1
Cross-site scripting	<script>alert(document.location);</script>
HTTP Response Splitting	KAU Hacker\r\nHTTP/1.1 200 OK\r\n...
Path Traversal	../../../../../../../../../../../../boot.ini
Command Injection	. & echo hello
XPath Injection	blah' or 1=1 or 'a'='a
LDAP Injection	(|(cn=*)%0A

Table 2.1: Malicious Input Examples for Web Application Vulnerabilities

An input validation vulnerability exists, if an attacker discovers that the application makes unfounded assumptions about the type, length, format, or range of input data and the input data is not validated. In this case, the attacker can then supply carefully crafted input that compromises your application and by exploiting the specific vulnerability a specific attack can be carried out on the web application.

To better understand how input validation vulnerability can act as a weak point in a web application, the architecture of a typical web application is shown in Figure 2.1. If a malicious input from the client, which is usually a web browser, is sent to the web application and it is used without validation, it can access unauthorized resources like database and files.

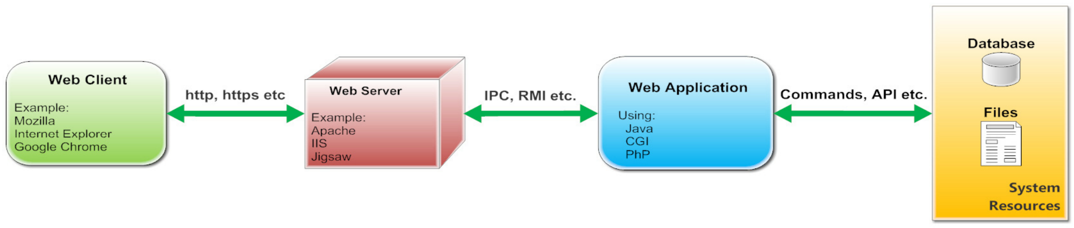

Figure 2.1: Architecture of typical web based application

2.5.3.1 SQL Injection

Constructing a dynamic SQL statement with user input and embedding it in the source code may allow an attacker to modify the statement's meaning or to execute arbitrary SQL commands. SQL injection occurs when a specially crafted user input is passed to sensitive SQL query execution functions without prior validation, which results in change of semantics of the SQL query. By exploiting an SQL Injection (SQLI) vulnerability a malicious individual can execute an arbitrary SQL query on the server.

Example Source Code

Listing 2.1 shows an example Java source code that contains an SQLI vulnerability. This code would allow an attacker to inject code into the query that would be executed by the database. The variable "custID" at line no.2 is being used to compose a query string at line no.5 and this query string is being executed at line no.11.

```
// Getting the customer ID
String custID = request.getParameter("id");

// Composing the query string.
String query = "SELECT * FROM accounts WHERE custID='" +
request.getParameter("id") + "'";

try {
// Executing the query.
Statement stmt = connection.createStatement ();
ResultSet resultSet = stmt.executeQuery(query);
}
```

Listing 2.1: Example of SQL Injection Vulnerable Code

Malicious Input

Listing 2.2 shows one of the possible malicious inputs. As the source code is taking the customer id as an input from the user, the user can craft his input and provide it as "**' or '1' = '1**" for carrying out an SQLI attack.

```
http://example.com/app/accountView?id=' or '1'='1
```

Listing 2.2: Example Malicious Input for SQL Injection Attack

Result

The above malicious input changes the meaning of SQL query to return all the records from the accounts table. As the query will be rendered by SQL Server as shown in Listing 2.3

```
SELECT * FROM accounts WHERE custID='' or '1' = '1'
```

Listing 2.3: Result of Malicious SQL Query Input

2.5.3.2 Cross-Site Scripting

"Cross-Site Scripting (XSS) attacks are a type of injection, in which malicious scripts are injected into otherwise benign and trusted web sites. XSS attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user" [31]. Sending unvalidated data to a Web browser can result in malicious code (usually scripts) execution in the victim's browser.

Example Source Code

Listing 2.4 shows an example Java source code that contains an XSS vulnerability. This code would allow an attacker to perform an XSS attack. It is possible in the given example source code to provide a crafted input to the parameter "CC" as this is being read from the request object. Here for performing an XSS attack the user can easily craft a malicious script and provide it as an input string.

```
....
(String) page += "<input name='creditcard' type='TEXT' value='"
    + request.getParameter("CC") + "'>";
....
```

Listing 2.4: Example of XSS Vulnerable Code

Malicious Input

Listing 2.5 shows a possibility of malicious input that can be crafted by the attacker. In the example source code above, if the value of CC is given as shown in the example malicious input below, the user will become the victim on an XSS attack. Here the user has crafted his input as script which will executed in the web application.

```
CC='><script>document.location=
'http://www.attacker.com/cgi-bin/cookie.cgi?foo='+document.cookie
</script>'
```

Listing 2.5: Example Malicious Input for XSS Attack

Result

The above malicious input causes the victim's session ID to be sent to the attacker's website, allowing the attacker to hijack the user's current session.

2.5.3.3 HTTP Response Splitting

There exist two conditions for an HTTP response splitting attack to occur in web application [31]. The first one is for the data origin, if the data is originated from an untrusted source which is most frequently and HTTP request and according to the second condition, if it is used, replaced or added in an HTTP response header and then sent to a web user without being validated for malicious characters. In this case unvalidated data would be written into an HTTP header, which will allow an attacker to specify the entirety of the HTTP response and it will be rendered by the browser to complete the attack.

Example Source Code

Listing 2.6 shows an example Java source code that contains an XSS vulnerability. This code would allow an attacker to perform an XSS attack. It is possible in the given example source code to provide a crafted input to the parameter "CC" as this is being read from the request object. Here for performing an XSS attack the user can easily craft a malicious script and provide it as an input string.

```
 ....
String author = request.getParameter(AUTHOR_PARAM);
 ...
Cookie cookie = new Cookie("author", author);
cookie.setMaxAge(cookieExpiration);
response.addCookie(cookie);
 ....
```

Listing 2.6: Example of HTTP Response Splitting Vulnerable Code

Malicious Input

As in the above example vulnerable code, the author parameter is being read from the request, the attacker can craft the input to split the response as shown in Listing 2.7.

```
“KAU Hacker\r\nHTTP/1.1 200 OK\r\n...” (A string containing CR and
LF characters)
```

Listing 2.7: Example Malicious Input for HTTP Response Splitting Attack

Result

As a result, the HTTP response will be split into several lines as per the given input. In the input, user include some carriage return and line feed characters that will actually split the response as shown in the output in Listing 2.8. In the similar way, different input can be crafted to achieve any other objective and attacking the HTTP response.

```
HTTP/1.1 200 OK
...
Set-Cookie: author=KAU Hacker
HTTP/1.1 200 OK
...
```

Listing 2.8: Result of HTTP Response Splitting Attack

2.5.3.4 Path Traversal

The path traversal vulnerability allows user to control paths used by the application. By crafting a malicious input the attacker may access the files that are actually meant to be protected. The files and directories outside the web root folder are the actual target of a path traversal attack [31]. Thus, the attacker can then access the files that are stored on a web server's file system. These files may include web application source code, configurations files and other system files.

Example Source Code

Listing 2.9 shows how the application deals with the resources in use. Typically, the paths to the resources present on the web server are used by the web application in the style shown in the listing below.

```
http://some_site.com.br/get-files.jsp?file=report.pdf
http://some_site.com.br/get-page.php?home=aaa.html
http://some_site.com.br/some-page.asp?page=index.html
```

Listing 2.9: Example of Path Traversal Vulnerable Code

Malicious Input

For an attacker to execute a path traversal attack, he can insert a malicious string as the variable parameter to access files located outside the web publish directory. The possible malicious inputs are shown in the Listing 2.10.

```
http://some_site.com.br/get-files?file=../../../../some dir/some
file

or

http://some_site.com.br/../../../../some dir/some file
```

Listing 2.10: Example Malicious Input for Path Traversal Attack

Result

As a result, the attacker would be accessing the files located outside the web publish directory. In the case shown in example the user will be able to access “some file” in the directory “some dir”.

2.5.3.5 Command Injection

Command injection is similar in nature to the other injection vulnerabilities. If a source code contains a command injection vulnerability, the attacker can exploit it execute commands by injecting some malicious code through an unvalidated input source. By exploiting this vulnerability, the attacker can inject and execute unwanted system commands with the same privileges and environment as the application has.

Example Source Code

Listing 2.11 shows a Java source code that contains a command injection vulnerability at line no.14 and line no.15. The source code is concatenating a user input from the command arguments to a command string that is being executed using the current runtime environment.

```
    ....
Runtime runtime = Runtime.getRuntime();
String[] cmd = new String[3];
cmd[0] = "cmd.exe" ;
cmd[1] = "/C"; cmd[2] = "dir " + args[0];
Process proc = runtime.exec(cmd);
    ....
```

Listing 2.11: Example of Command Injection Vulnerable Code

Malicious Input

For carrying out an attack, the attacker can provide any malicious input as command arguments. For example in Listing 2.12, the user can give a string containing a system command i.e. “echo”.

```
“. & echo hello”
```

Listing 2.12: Example Malicious Input for Command Injection Attack

Result

As a result, the dir command will list the contents of the current directory and the echo command will print a friendly message. However, the user could have used any command other than “echo” to execute it on the system.

2.5.3.6 XPath Injection

An XPath injection occurs when an XPath query is constructed from user input without validating it. This vulnerability is similar to the SQL injection in nature, as the attack can occur if the user supplied input is used to construct an XPath query to get the XML data. This vulnerability enables the attacker to get the structure of XML data, or access the data itself that is stored in the XML file, where in some cases this data can be some authentication details.

Example Source Code

Listing 2.13 shows an example Java source code that contains an XPath Injection vulnerable code. This code would allow an attacker to perform an XPath Injection attack. It is possible in the given example source code to provide a crafted input to one of the request parameters including “Username” or “Password”.

```
    ....
String FindUserXPath;
FindUserXPath = "//Employee[UserName/text()='"
    + Request("Username") + "' And Password/text()='" + Re-
quest("Password") + "']";
    ....
```

Listing 2.13: Example of XPath Injection Vulnerable Code

Malicious Input

As in the above example vulnerable code, the author “Username” or “Password” parameters can be sent in the request by crafting them maliciously. The Listing 2.14 shows one of the possible malicious string that can be passed to perform and XPath injection attack.

```
Username: blah' or 1=1 or 'a'='a
Password: blah
```

Listing 2.14: Example Malicious Input for XPath Injection Attack

Result

As a result, the attacker gets an XML node selected without knowing the username or password. The password part becomes irrelevant, and the UserName part will match ALL employees because of the "1=1" part. The input given by the user will change the meaning of the XPath query and it will be interpreted as shown in Listing 2.15.

```
//Employee[UserName/text()='blah' or 1=1 or 'a'='a' And Pass-
word/text()='blah']
```

Listing 2.15: Result of XPath Injection Attack

2.5.3.7 Lightweight Directory Access Protocol Injection

As it is obvious from the name, the Lightweight Directory Access Protocol Injection (LDAPI), is an injection type vulnerability. This also depends on the unvalidated user input and behaves like SQL injection vulnerabilities. So the techniques similar to SQLI can also be applied for LDAPI. This attacks can be used to exploit web based applications that construct LDAP statements based on user input.

Example Source Code

The source code in Listing 2.16 shows an example in Java that contains an LDAPI vulnerability. This code would allow an attacker to inject code into the LDAP query that is being printed on the console in this example. However, in other scenarios, this is possible that some other sensitive operations are being carried out on the basis of this constructed LDAP query.

```
     ....
<input type="text" size=20 name="userName">Insert the
username</input>
.........
String ldapSearchQuery = "(cn=" + $userName + ")";
System.out.println(ldapSearchQuery);
     ....
```

Listing 2.16: Example of SQL Injection Vulnerable Code

Malicious Input

For different possible attacks, different crafted inputs can be given to the parameter "userName" in the above example source code. However, two of these possibilities may include:

1. User inputs the userName as "*" in the input field.
2. User gives "imran9pk)(|(password=*))" as input in the input field.

Result

The result of the malicious inputs in each of two cases, as mentioned in the example inputs would be as below:

1. The system may return all the usernames on the LDAP base.
2. It will generate the code revealing imran9pk's password by generating the code:

   (cn = imran9pk) (| (password = *))

2.6 Mobile Agents

Mobile agents are actually software agents and can be easily defined by the help of software agents. Mobile agents are programs that can move through a network under their own control, migrated from host to host and interacting with other agents and resources on each host [32]. However, from several years, the debate for defining the software is continued and still the research community has not agreed upon a common definition for software agents. There exists a common understanding about certain minimal features for a software entity to be qualified as an agent. These features include autonomy, social behavior, reactivity and proactivity [33].

Several works in the literature exist which are directed towards agent-based testing of web applications. Each of them support different test types among which Huang et al. [34] presents a framework that supports the security tests but is not based on the distributed architecture. It takes two types of attacks into account namely SQL Injection and Cross Site Scripting (XSS). Another work by Paydar et al. [35] supports the security testing along with some other non-functional tests. Also the architecture is distributed in nature which is based on the idea of mobile code i.e. Mobile Agents.

2.7 Agent Platforms

An agent platform is an environment responsible for providing Multi-Agent Systems (MAS) with necessary means for management, communication, negotiation and coordination among the agents. There are two types of agent platforms, where one is for the implementation and deployment of multi-agent systems which include JADE, FIPA-OS and Zeus. Whereas, the other type supports the execution of agents and their secure traveling on the network which include TACOMA, Tele-script and Aglets.

For the implementation of our agent based security testing tool, we have used the JADE [36] framework.

2.7.1 JADE Framework

JADE is a software development framework implemented fully in Java language aiming at the development of multi-agent systems and applications which are designed as agent based systems. JADE provides standard agent technologies and offers the developer a number of features which simplify the development process. JADE is a distributed agent platform which enables the agent communication through message passing and it uses standard interaction protocols for carrying on the communication with the software agents. Also, as being the platform for the agents, JADE is responsible for managing the agent tasks and behaviors.

Chapter 3

Related Work

While detection and prevention of security errors in web based applications has taken much attention in the last decade where static analysis has been used from long time, applying dynamic analysis and integrating it with the static analysis in the form of mobile agents is new. Different approaches exist to detect the security vulnerabilities but these can be broadly classified into two categories namely static analysis and dynamic analysis. Under these categories, several variants are used in the form of different solution and tools which are reviewed in the rest of this chapter.

3.1 Static Analysis

The static analysis technique has been used in the solutions requiring analysis from long time, whereas using it to detect security problems through the source code was introduced in ITS4 [37]. ITS4 is a static analysis tool that internally tokenizes the target source code performs lexical analysis on that. The tool basically scans the source code and look for syntactic matches where it uses a vulnerability database and reads it at the runtime and compares the rules with the target source code. Whereas the database containing the rules can be modified for enhancing the security knowledge of the tool. Flawfinder [38] and RATS [39] works in the similar fashion as the ITS4. The target source code is syntactically searched for a predefined set of vulnerable methods. The difference between them is based on the language support where ITS4 and Flawfinder can work on C and C++ codes however, RATS can analyze languages more than that including Python, PHP and Perl. These tools perform very simple but sound static analysis as per their rules base but the number of false positives is very high.

A little more advanced tools like LCLint [40] and SPLINT [41] where latter is the extension of the earlier, are capable of performing several lightweight static analysis. LCLint detects null pointer dereferences, dead storage, memory leaks, and other programming issues by using the source code annotations that are provided by the user. The SPLINT [41] focuses on the secure programming by setting up models for the program control flow and using several lightweight static analysis.

BOON [42] targets the buffer overflow vulnerabilities in C which occur due to the use of string manipulation function. It models all the standard library functions that are used for string manipulation, with the amount of memory allocated and the number of bytes currently used for performing an integer range analysis. For performing the analysis integer range constrains are generated for each statement in the program and after solving the constraint system warning is reported if a statement violates the constraints.

A static analysis tool called ARCHER [43] is meant for checking memory access errors. It does the interprocedural analysis using the call graph and actually operates on the control flow graph. Another framework called cqual [44] uses user-defined type qualifiers. Cqual checks the format string vulnerabilities in C programs by performing type checking on the program.

Model checking approach is used in the tool UNO [45]. It is scans three commonly occurring defects in C language programs which includes the use of uninitialized variables, null pointer references and out of bounds array indexing.

Vulncheck [46] tracks the tainted data by using the data flow analysis and detects the vulnerabilities like buffer overflows, format string vulnerabilities, memory leaks and insecure C functions. The analysis performed by the Vulncheck is flow-sensitive and follows the statements order in the program. The analysis performed by the Vulncheck is intra-procedural only which means it do not look for the vulnerabilities that depend on several functions.

Another model checking static analysis is performed in MOPS [47]. The rules are defined as temporal safety properties and these rules can also be defined by the users themselves. However, the users should be familiar with the finite state automata to define these rules. These rules are then used to perform the analysis and check for the order of security relevant operations in the source code by using a push down automata model. It uses the model checking to determine whether a path to the state exists which violates the security. The tool takes only the control flow into account and neglects the data flow and performs inter procedural analysis. This tool can be used by the developers and code reviewers to detect file access race conditions, vulnerable statements and privilege management errors.

Another approach by Livshits [48] performs an interprocedural taint analysis and detects the security vulnerabilities in Java programs. The order of the statements in the program are not taken into account as the analysis performed by their approach is flow insensitive. Their static analysis approach is based on the pointer analysis for finding the security vulnerabilities in Java based Web Applications. The taints are propagated in the program and the dataflow analysis is used to find sinks that uses the tainted objects that are derived from the sources. The vulnerabilities targeted by their approach include SQL injection, XSS, HTTP response splitting, path traversal, and command injection attacks.

Another static analysis tool called Pixy [49] is developed for detecting the XSS vulnerabilities in the PHP web applications. This tool uses a taints analysis based on the data flow analysis. A control flow graph is constructed for doing the data flow analysis. Once a taint is detected, a pointer analysis is used to find out if any other variables pointing to the same memory locations. If this happens, these other variables are also marked as tainted. So, such variables are marked as tainted which originate from untrusted PHP functions and they reach critical PHP functions.

An approach for detecting the SQL injection vulnerabilities is introduced by Xie [50] in the form of three tier architecture where intra-procedural and inter-procedural information is computed by a backward dataflow analysis. One limitation that exists is the missing support for the recursive calls as recursive function calls are not treated as ops. Also the vulnerability results only give the source of tainted values which actually slows down the process of manual human inspection and if any complex vulnerability is encountered it becomes more complex to do the manual post inspection.

Griffin software security project [51] introduced a plugin for eclipse called Lightweight Analysis for Program Security in Eclipse (LAPSE). This tool also uses the taint analysis approach and detects source and sink methods in the Java web based applications. It first looks for all the possible sources where the sources are methods that are used to take input form the untrusted sources. After the sources, all the sink methods are found and then tool tries to find a path from source to any of the sink methods. If any such path is found, a vulnerability is reported. All the source and sink methods are placed in an XML data structure which can be modified and extended.

Checkstyle [52] is developed as a style checker and it checks the code against defined patterns. This tool performs dataflow analysis on the AST for finding the problems in the source code. The coding errors like security vulnerabilities cannot directly be detected by it, but a Check class that is used for all the checks, can be extending according the requirement and then this extended check class can be used for detecting the security vulnerabilities in the source. The tool first builds an AST and then visits the AST using the visitor design pattern and applying the custom check by calling it while traversing the AST. This tool targets the source codes written in Java and requires custom checks to be written for required errors by extending the already available Check class.

Theorem proving has also been used for performing the static analysis on the Java source code. One of the tool using such technique is called ESC / Java (Extended Static Checker for Java) [53].

One of the static analysis tools which perform static analysis on the bytecode instead of the source code directly, include a tool called FindBugs [54]. This tool checks the code against numerous types of bug patterns, among which one of the category is security. In this category, different rules exist for finding the security vulnerabilities in Java applications. Also, for extending the scope of the tool, own rules can also be defined for checking more vulnerabilities. FindBugs detects the vulnerabilities or other bugs by comparing the binary codes against the error pattern detectors. It uses the Bytecode Engineering Library (BCEL) for analyzing the Java bytecode statically. For completing the static analysis, a symbol table is built and dataflow analysis is performed in forward and backward directions on the control flow graph built from the target bytecode.

SAFELI [55] is a proposed Static Analysis Framework in order to detect SQL Injection Vulnerabilities. SAFELI framework aims at identifying the SQL Injection attacks during the compile-time. It does a White-box Static Analysis and a Hybrid-Constraint

Solver are used in this framework. White box static analysis is done by considering the byte-code and dealing mainly with strings. However, an efficient string analysis tool is implemented to deal with the Boolean, integer and string variables for the Hybrid Constraint Solver. The constraint solver serves two task which are to decide satisfiability of path constraints and to find out the initial values of input variables that lead to the breach of database security. The implementation of SAFELI was done on ASP.NET web applications and it detected the vulnerabilities that were ignored by other black box vulnerability scanners. Another work by Thomas [56] suggest an automated prepared statement generation algorithm to remove SQL Injection Vulnerabilities. Based on the experimental results, their prepared statement code was able to successfully replace 94% of the SQLIVs in four open source projects. However, the experiment was conducted using only Java with a limited number of projects. Hence, the wide application of the same approach and tool for different settings still remains an open research issue to investigate.

Another famous tool known as PMD [57] is a static analysis tool. It performs static analysis on the Java source codes on the basis of predefined rules for finding the vulnerabilities. The development purpose of this tool was to detect other programming mistakes, whereas it can also be tailored for detecting security vulnerabilities [58]. The rules known as bug patterns can be added to its rule base. These rules are applied by the tool on the AST built from the source code using a plugin of JavaCC called JJTree. Each of the new bug patterns are to be implemented as a class which extends the abstract Java rule class. In this class the methods are provided which are called when an AST is being traversed. The AST is traversed by using the visitor design pattern like in many other tools.

In Table 3.1 a collection of tools has been listed along with their target language and the static analysis technique they use.

Tool/Feature	Target Language	Static Analysis Technique
ITS4	C/C++	Lexical Analysis/Pattern Matching
Flawfinder	C/C++	Lexical Analysis/Pattern Matching
RATS	Multiple	Lexical Analysis/Pattern Matching
Splint	C/C++	Control Flow Analysis
BOON	C	Model Checking
ARCHER	PHP	Control Flow Analysis
cqual	C/C++	Type Checking
Vulncheck	C	Data Flow Analysis
MOPS	Multiple	Model Checking
FindBugs	JAVA	Lexical Analysis/Pattern Matching + Data Flow Analysis
Checkstyle	JAVA	Lexical Analysis/Pattern Matching
PMD	JAVA	Lexical Analysis/Pattern Matching
Jlint	JAVA	Lexical Analysis/Pattern Matching + Data Flow Analysis
Parfait	C/C++	Data Flow Analysis
Astree	C/C++	Abstract Interpretation
Frama-C	C/C++	Abstract Interpretation
UNO	C/C++	Model Checking
Bandera	JAVA	Model Checking
Java Pathfinder	JAVA	Model Checking
ESC	JAVA	Theorem Proving

Table 3.1: Sampling of Static Analysis Tools

3.2 Dynamic Analysis

In the literature several dynamic data flow analyses have been proposed, researched and used in the software security solutions and tools. In a dynamic taint analysis, the program is executed and those computations and method calls are searched which are affected the unvalidated user input. Different works targeting the security vulnerabilities problem follows this approach. One of these works include a dynamic taint analysis approach by Haldar [59]. This approach targets the Java web applications and looks for XSS, Cookies Poisoning and Command injection attacks at the runtime. One of the approaches [60], prevents the SQL injection attacks during the runtime by building a parse tree of the SQL statements before including the user input and after including the user input in the SQL statement and comparing both of them. By comparing the before and after user input parse trees, it is decided whether the structure of the statement is same and attack is attempted or not. One other similar approach based on the parse trees building at the runtime is used in the tool called CANDID [61]. It uses its runtime analysis for preventing the SQL injection attacks. The attacks are prevented by recording the sequence of SQL commands and replacing the inputs in these commands with 1s and then building the parse tree. If the parse tree differs from the original parse tree, the query is preventing from execution.

An approach from Boyd [62] uses a functionality provided by Java for preventing the SQL injection attacks. They use the PreparedStatement API available in Java and forces the SQL queries to be containing only string or numeric literals. Also the SQL keywords are randomized, so that they could not be guessed by the users however, this is the limitation in this approach also, as it would be compromised it user guesses the randomization key successfully.

Another dynamic taint analysis approach by Chang et al. [63] is targets the C programs and looks for the command injection attacks and format string attacks. In this approach, a data flow analysis is integrated at the compile time using a small library that tracks the taints throughout the program during execution for detecting the vulnerabilities that are caused due to inputting the untrusted data.

The dynamic taint analysis is a common approach to follow for preventing the attacks at the runtime time. Several works which include [64] and [65] used this form of analysis. The dynamic taint analysis is actually influenced by the Perl's taint mode. Also, one other way to dynamically prevent the attacks from happening is the use of wrappers [66]. The wrapper to the program will filter out the malicious input values which will eventually prevent malicious input to reach the actual program and the security vulnerabilities would not be exploited.

The dynamic taint analysis is not only used for Java, but Salvatore [67] also used it for tracking the taints information at the character level in the PHP programs. In this proposed technique, the SQL query is tokenized and checked for the existence of any tainted values in it. Similarly, Wasp [68] uses tainting technique for Java by providing the bytecode instrumentor and tainting the strings.

3.3 Integration of Static and Dynamic Analysis

There are pros and cons for both the static and dynamic analysis techniques where the static analysis is able to do high code coverage with low accuracy and dynamic analysis is opposite to that. To neutralize their cons and maximize their pros, the integration of these two techniques has been the subject of this project and similarly several other researches in the past have used this concept in their solutions. A very simple example for the integrated technique is the technique used to prevent the XSS attacks on the client side, by combining the static and dynamic analysis in a web browser [69].

The work by Lucca et al. [70] is based on identifying Cross-Site Scripting vulnerabilities in web applications. It presents and approach which is combination of Static and Dynamic analysis where static analysis is supposed to detect potential vulnerabilities and then the dynamic analysis will help in detecting the actual vulnerabilities. To prevent the XSS attack one of the recommended solution is to disable the scripting languages in the bowser however this problem should be addressed by the developers instead on end users. Another option suggests to use the input validation functions after each input but this will result in an overhead as all the inputs might not affect the output data which will not cause XSS. This work proposed an approach to analyze only the input data which affects the output data for which it exploits both static and dynamic analysis. They used some predicates to define some rules by applying them to the Control Flow Graph (CFG) of the server page for assessing its vulnerability. By using the predicates in some conditions the vulnerabilities are characterized as Potentially Vulnerable (PV), Vulnerable with respect to v (V) and not vulnerable (NV). For the dynamic analysis, output of the static analysis is exploited by submitting only those pages which were found vulnerable in the static analysis. For the dynamic analysis the author defines a set of XSS attack strings and for each string it executes each vulnerable server page by giving the attack string as input to each vulnerable field of that page, after which the attack consequences are checked. To test the effects of the attack in the dynamic analysis it might be difficult when the output/malicious data is not provided to the user but stored in the database. Thus to observe the effects of XSS WATT (Web Application Testing Tool) has been used which takes the input from the XSS test case generator module and the results of test case execution are checked to assess the success of the attack.

An integrated technique is used in a tool Saner [71]. It detects the sanitization routines in a program with a static analyzer based on the already existing tool called Pixy [49]. After the static analysis is done, the dynamic analysis is used making the tool more sound and complete by checking if the detected sanitization is correct and complete.

Amnesia [72] integrates the static and dynamic analysis and used to prevent the SQL injection attacks at the runtime. The static analysis is performed in this tool by building a model of valid SQL queries and then in the dynamic analysis the queries generated at the run time are checked against the statically built model that whether these runtime queries comply with the statically built model.
One of the integrated techniques is proposed for detecting the security vulnerabilities in the PHP based web applications. This technique is used by a tool known as WebSSARI [73]. The static analysis of WebSSARI constructs the Abstract Syntax Tree, Control Flow Graph and uses then to track the state of variable in the program with the help of a symbol table. The path between the taint values and dangerous functions is identified and the next part is done for the runtime prevention and detection of attacks. Specific instrumentation code is inserted based on the static analysis results and this code performs checks and prevents the security attacks during the application runtime.

3.4 Other Security Techniques

As per the need of security, different technologies got developed to address the problem of web application vulnerabilities. As discussed in section 2.22.2 above, some of the commonly used solutions for testing web application security include runtime monitoring, penetration testing, intrusion detection, client-side protection and proper system designing keeping security in mind. In addition to that manual code reviews, code auditing and use of application firewalls can prove to be a good candidate for tackling the security problem. Enforcing specific security policies through the use of automated tools is also used as a security solution. SASI [74] implements the security policies by the use of security automata to prevent the memory access outside the allowed address space. JFLow [75] imposes the information flow policies by applying the information flow analysis and it is made sure at the runtime that there is no inadequate information flow.

Chapter 4

JAVA Web Application Security Tester

In this chapter we have explained the rationale for our developed technique and tool. We have explained our developed technique for detection and prevention of security vulnerabilities in web applications. Also, the architecture of our agent based security testing tool i.e. Java Web Application Security Tester (JWAST) has been discussed, where the high level architecture as well as the low level architecture of the proposed tool is presented. The design of our static analysis and dynamic analysis technique and their integration is also presented in this chapter.

4.1 Rationale

The rationale behind the JWAST is precisely to develop an agent based security testing tool based on the integrated technique that we built, so that we can test and compare our technique with other existing techniques.

As discussed in Chapter 3, a number of other tools and techniques exist that are meant for software security testing. However, we have tried to cover some gaps by designing and developing our own technique and tool. Our work can be rationalized by considering the following points:

1. A huge amount of services and operations are being deployed in the World Wide Web which are typically communicated through web browsers and people rely on this communication. Pressing need here is for a reliable web application security testing tool.

2. Not only the use of web applications is increasing, but also the number of attacks and attackers are increasing where the attack types are getting more and more sophisticated. This requires the security testing tools to evolve and use vast techniques for controlling these attacks.
3. The existing security testing tools are based on works ranging from commercial to open source and academic to non-academic researches and each of these tools target the security problem in a specific manner differing in the underlying technology and techniques used. A new security testing tool can be developed by making use of other technologies and techniques to detect and prevent other or more number of vulnerabilities in different manner.
4. The two commonly used techniques for software security testing include static analysis and dynamic analysis. The existing works make use of either static analysis techniques or the dynamic analysis techniques in isolation. However, only few of them integrate the two techniques in their proposed solutions. By integrating a new optimized static analysis technique and a dynamic analysis technique, a new technique can be built for developing a new security testing tool.
5. Each of the current security testing tools target variable number of vulnerabilities and also the classes or categories of vulnerabilities that they target are different. In a new security testing tool more number of vulnerabilities and different classifications of the vulnerabilities can be covered for security testing.
6. One other difference for the existing techniques is the target language that they operate on the form of input that they take for performing the test. A new developed technique by targeting the source code of the JAVA language for

security testing can differentiate the solution with many other already existing works.

4.2 The Testing Methodology

As we have explained different security testing techniques that can be used to test the software programs for known or unknown security vulnerabilities in Chapter 2. This section presents the technique that we built for the purpose of security testing of web based applications.

Our introduced tool called Java Web Application Security Tester (JWAST) is a static and dynamic testing tool which is based on our integrated static and dynamic analysis technique. The basic idea behind our technique is the use of static analysis technique and integrating it with a dynamic analysis technique to increase the detection capability of our tool and also enable our tool to prevent the attacks from happening at the run time.

We implemented the tool JWAST which enable the programmer to test the web based applications written in Java. This tool applies static and dynamic analysis techniques on the source code sequentially and the details for these techniques are explained in the next sections.

4.2.1 Static Analysis Technique

The tool starts by the code analysis process where at first the static analysis is performed. The static analysis operates on the Java source code files where it analyzes each file for detecting the specified vulnerabilities. The vulnerabilities are specified by the security rules which behave as the security knowledge for our static analysis technique. The static analysis starts by lexically analyzing the source code, for which the source code is broken into tokens. This task is performed by the lexical analyzer agent which gives the produced token stream to the parser agent as an input. The parser agent consumes the input token streams and generates an Abstract Syntax Tree (AST). This AST is actually used as a model, on which the actual static analysis is performed by the imports analyzer agents and taints analyzer agents. The imports analyzer agent is guided by the predefined security rules and the generated AST is given as an input to it. The imports analyzer agent generates the vulnerable imports which further guide the taints analyzer agents along with the same predefined security rules. The vulnerable imports are passed on to the taints analyzer agent for completing the analysis process. Finally, based on the generated AST, security rules and vulnerable imports knowledge, a taints analysis is performed which propagate the taints from source methods to sensitive sink methods and if a path is established between the taint source method and a sensitive sink method, and no validation is performed between this, a vulnerability is reported which contains the vulnerability type, location and other required information which is further used by the dynamic analysis for completing the security testing process.

4.2.2 Dynamic Analysis Technique

Once the static analysis is completed, the next step is to perform the dynamic analysis on the web application. The dynamic analysis carries out the testing process by the use of instrumentation technique. The instrumentation approach is based on the idea that, the attacks occurring due to the input validation vulnerabilities can be handled by adding the validation to the source code by instrumenting the original source code with the pre-defined instrumentation templates. Therefore, the instrumentation code would perform the validation on the input given at the runtime, as a result of which the attacks would be stopped from being carried out and also the attempt for an attack can be reported during the web application’s runtime.

To do this, an automated dynamic analyzer agent generates the instrumentation code based on the instrumentation templates that contains the specified templates for each target vulnerability type. Later on, the dynamic analyzer agent also inserts the generated instrumentation code into the original web application code automatically. For inserting the instrumentation code, the locations are extracted from the results produced by the static analyzer agent. As, the instrumented source code, which is actually combination of the original source code and the instrumentation code, is executed the runtime attacks are prevented as well as reported to the user. The detailed architecture of the dynamic analyzer agent is shown in **Error! Reference source not found.**.

4.3 Java Web Application Security Tester Architecture

The architecture of our proposed tool, Java Web Application Security Tester (JWAST), for the security testing of web based applications is based on several independent analyzer agents, each of which is responsible for a part of the whole analysis process. Each agent takes an input either from a specified external source or from the output of another agent. All of these inputs and outputs have specified formats, on the basis of which other agents perform the required steps to complete the overall analysis meant for security testing. A high level architecture diagram of JWAST has been shown in **Error! Reference source not found.**.

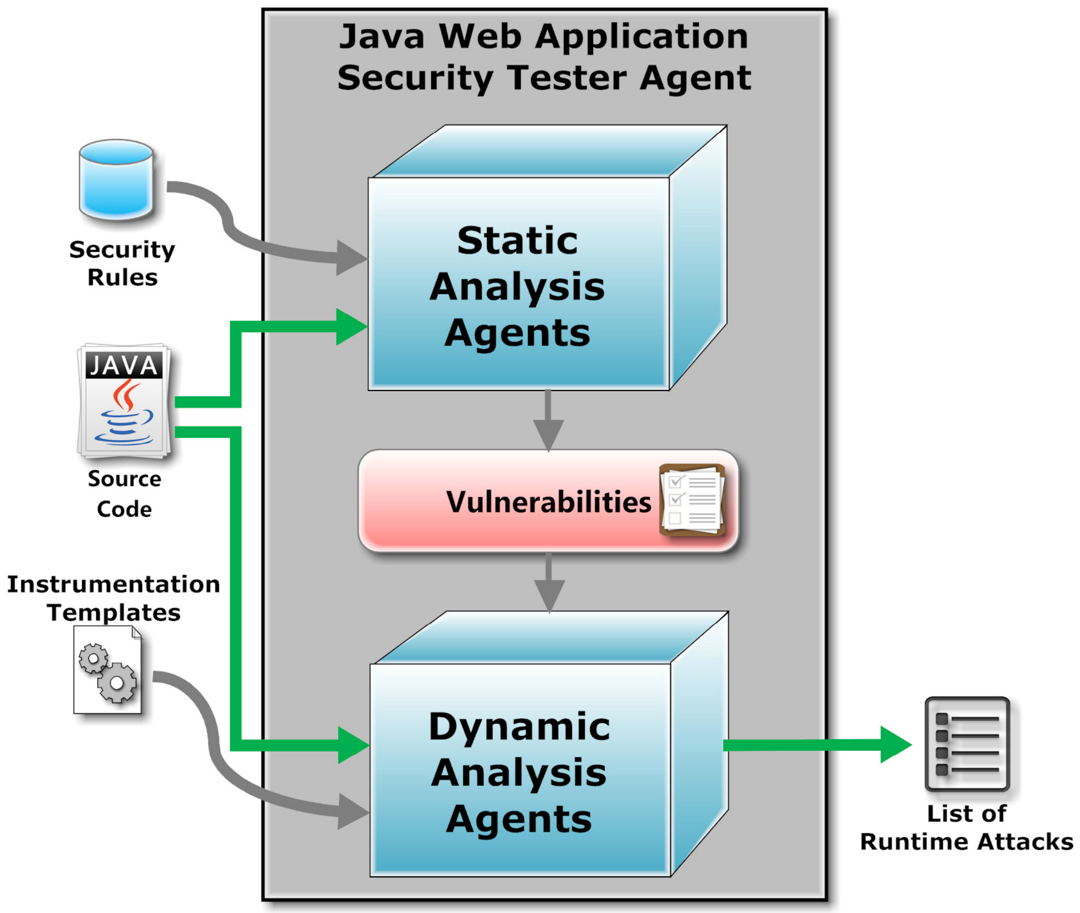

Figure 4.1: High Level Architecture of Java Web Application Security Tester

Based on the architecture presented in the Figure 4.1, the JWAST operates on the source code. An abstract flow during the testing process goes as described below:

1. Source code of the web based application written in Java is given as input to the static analyzer agent.
2. The static analyzer agent performs a static analysis based on our technique and outputs list of vulnerabilities containing the required information for the next phase.
3. The list of vulnerabilities generated during the static analysis are inputted to the dynamic analyzer agent. Also the instrumentation templates and the source

code that is to be instrumented for the runtime analysis is given to the dynamic analyzer agent.

4. The dynamic analyzer agent instruments the source code of the web application by inserting the instrumentation code in the original source code on the locations reported by the static analyzer agent.
5. The instrumented web application is compiled and executed. During the execution the runtime attacks are prevented and a list of runtime attacks is generated and presented as a final output of the tool.

4.4 The Architecture in Detail

As in the previous sections, we have presented the high level architecture of our tool. This section presents the low level architecture of our tool in details where each agent's architecture is presented and described in detail.

4.4.1 Static Analysis Agent Architecture

Static analysis agent is the entry point of our testing process. The static analysis agent is mainly responsible for performing the static analysis part of our technique. Our static analysis technique is based on the taints analysis where the taints flow is tracked in the program to check whether the taints i.e. the unvalidated inputs reach the predefined sink methods. The static analysis agent takes the Java web application source code as input and produces a list of vulnerabilities as the output. The static analysis agent utilizes other agents to complete the analysis.

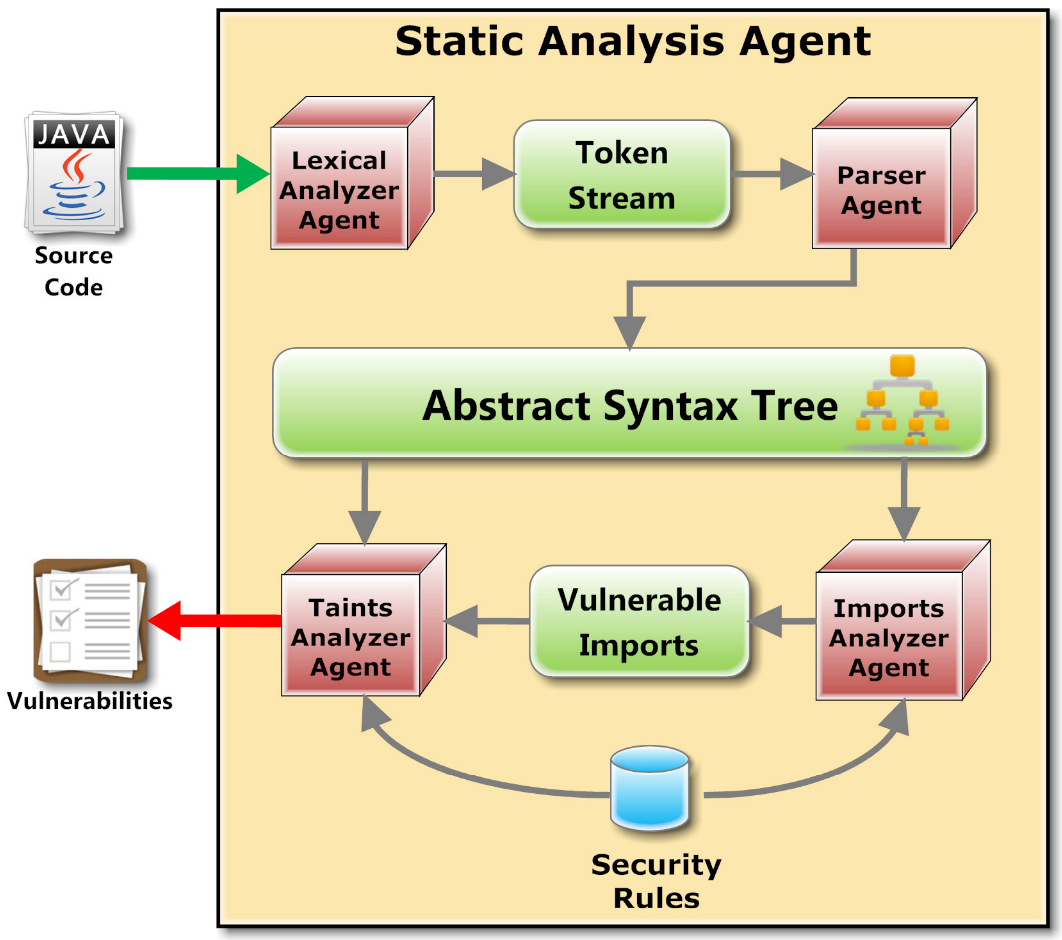

Figure 4.2: Static Analysis Agent Architecture

The detailed operation of the static analysis agent is shown in the Figure 4.2 and can be understood by the following operations.

1. Java source code of the web application under test is provided to the lexical analyzer, where it is tokenized and the tokens are passed on to the parser agent.
2. The source code tokens are consumed by the parser agent which parses them to generate an abstract representation of the source code. As a result, an Abstract Syntax Tree (AST) is generated.

3. The imports analyzer agent performs analysis on the AST to find vulnerable imports in the given code. These vulnerable imports are generated on the basis of the security rules that are also inputted to the imports analyzer agent.
4. The taints analyzer agent takes the vulnerable imports generated by the imports analyzer agent and uses the already generated AST to perform the taints analysis based on the predefined set of security rules.
5. The taints analysis performed by the taints analyzer agent finally produces the list of vulnerabilities. This list of vulnerabilities contains detailed vulnerabilities information including the locations and types of the vulnerabilities and later on, this list is further consumed by the dynamic analysis agent.

4.4.1.1 Lexical Analyzer Agent Architecture

The architecture of the lexical analyzer agent is shown in Figure 4.3. The lexical analyzer agent performs the lexical analysis on the given source code and the source code is tokenized. First the input source code file is converted into character stream and then the token manager generates the token stream. The tokenizing of the Java source code is performed on the pre-specified Java regular expression grammar. The lexical analysis is actually inspired by the basic functionality provided by the compilers. The compiles also perform a basic lexical analysis on the source code to start the compilation.

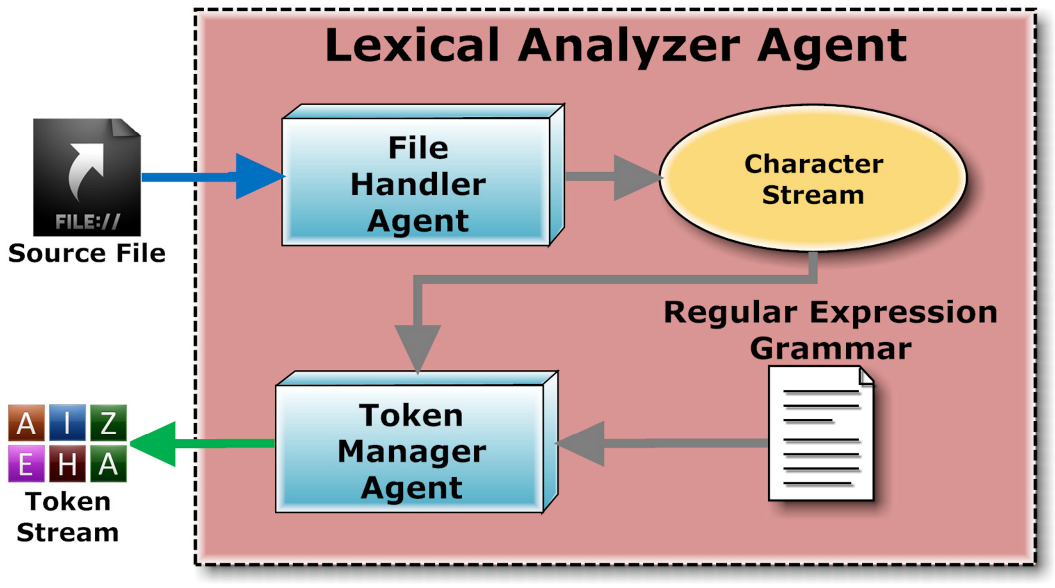

Figure 4.3: Lexical Analyzer Agent Architecture

4.4.1.2 Parser Agent Architecture

The next step after the lexical analysis in compilers is parsing. The token stream is used to parse the source code. The token stream produced by the lexical analyzer is consumed by the parser agent which is shown in Figure 4.4. The parser agent is responsible for parsing the input tokens stream which is done by the use of a context free grammar that is given in a predefined format written for parsing the Java programs. The parser agent matches the tokens in token stream with the production rules in the grammar. If the symbols represented by each token match, the parse tree is formed incrementally. By abstracting the details of the grammar, the Abstract Syntax Tree (AST) is formed.

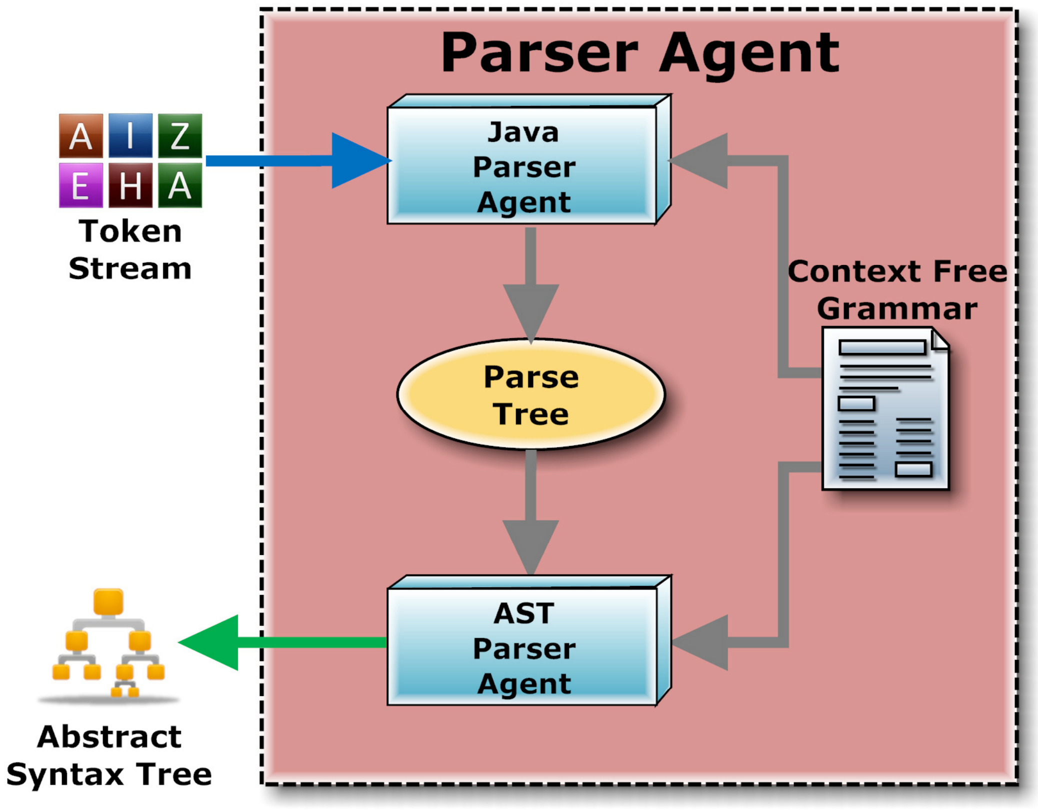

Figure 4.4: Parser Agent Architecture

4.4.1.3 Imports Analyzer Agent Architecture

Imports analyzer agent takes the input in the form of AST. The imports extractor agent simply extract the imports from the AST nodes and passes on to the vulnerable imports scanner agent. The scanner agent scans all the imports using the knowledge gained from the security rules and list of vulnerable imports along with the source code file is reported to the next agent. The architecture diagram for imports analyzer agent is shown in Figure 4.5.

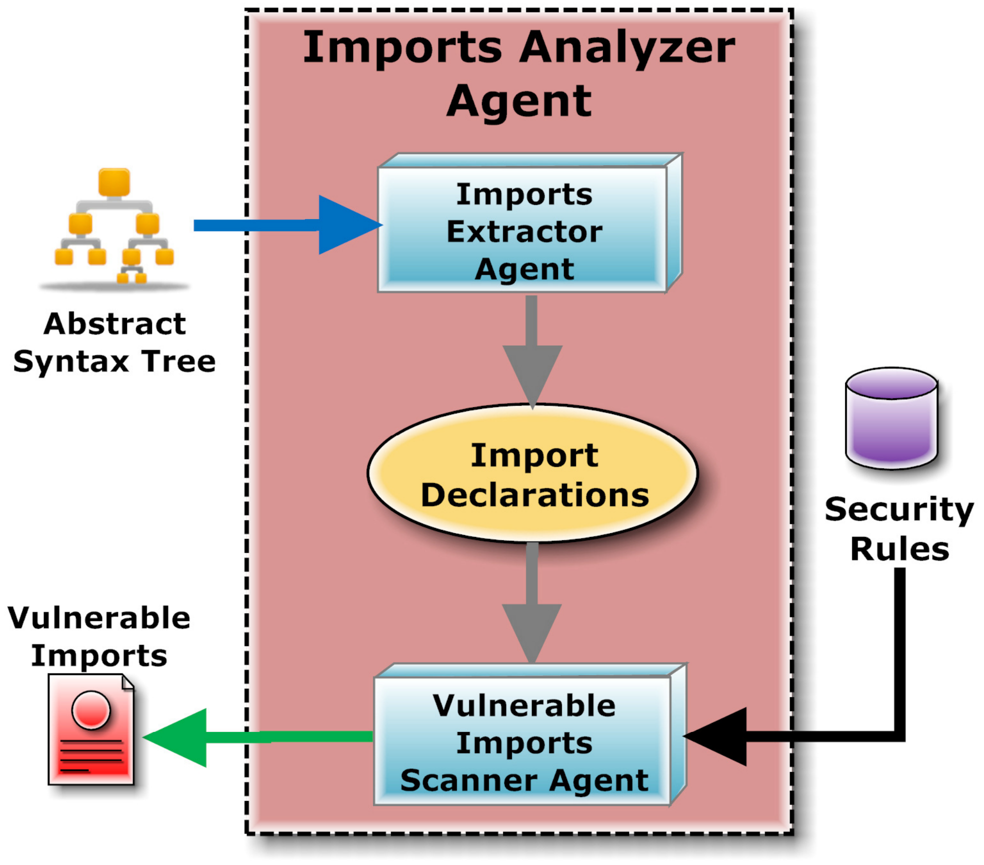

Figure 4.5: Imports Analyzer Agent Architecture

4.4.1.4 Taints Analyzer Agent Architecture

The static analysis agent mainly depends on the taints analyzer agent. The outputs generated by the previous agents are used by the taints analyzer agent for completing the analysis. The taints analyzer agent is based on the data flow analysis. The taints analysis performed by the taints analyzer agent is based on the data flow analysis which is used to find out the way how data moves through a program. The generated AST is visited (traversed) to detect the generation and use of taints (vulnerable inputs) in the program.

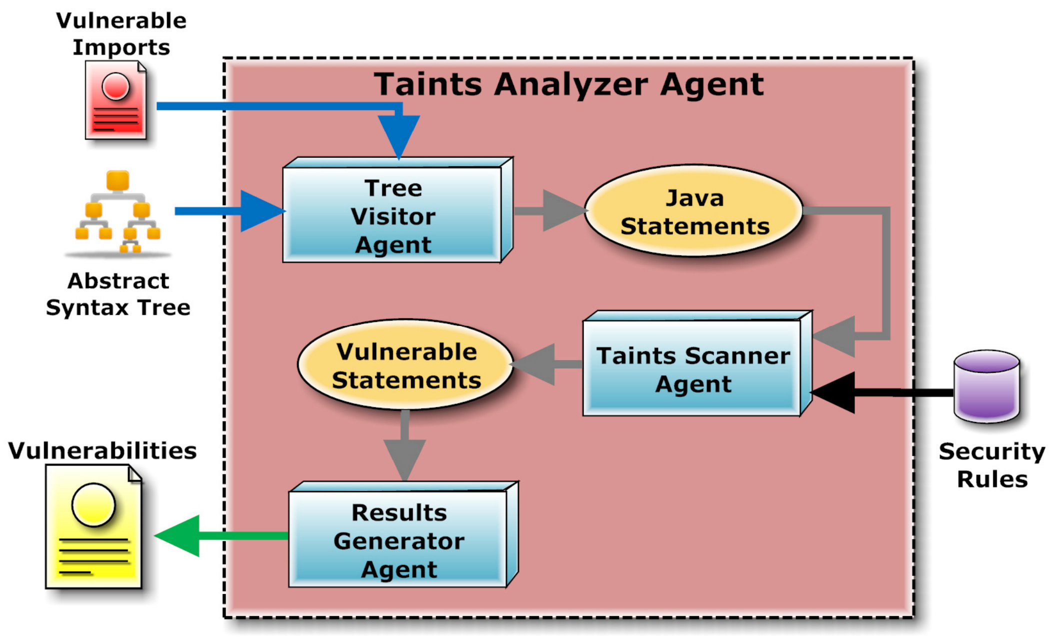

Figure 4.6: Taints Analyzer Agent Architecture

As we have shown in Figure 4.6, the tree visitor agent consumes the AST and vulnerable imports and visits the whole AST guided by the vulnerable imports. As a result of tree visiting, Java statements are generated, which are then given to the taints scanner agent. The taints scanner agent is guided by the security knowledge in the security rules to identify the vulnerable statements. Once the vulnerable statements are identified, the results generator produces the list of vulnerabilities by confirming the flow of taints from the source methods to the vulnerable sensitive sink methods.

4.4.1.5 Security Rules

Different tools have their own definitions for the security rules in differently specified formats. In our tool, JWAST, we have defined the rules in an XML format where these are divided into source rules and sink rules. Snippets of the source rules and sink rules are shown in Listing 4.1 and Listing 4.2 respectively. The source rules define the methods and sources that can be used in Java for taking the input from the user. These rules help us identify and mark the user inputs as taints. Further, the sink rules specify sensitive sink methods. Sink methods are the methods which if given an unvalidated user input i.e. a taint, then vulnerability is created and possibility of attacks execution is created. The security rules drive the imports and taints analysis performed by the static analysis agent. The security rules can also be stored in a database and then used by the tool, but the main reasons for using and XML data files include, simplicity of managing and storing the rules, scalability provided by the XML files and the small size that an XML file takes as compared to any database file.

```
<source id="java.lang.System.getEnv(String)">
    <package>java.lang</package>
    <class>System</class>
    <method>getEnv</method>
    <category>Untrusted Source</category>
</source>
```

Listing 4.1: A Source Rule from the Security Rules

```
<sink id="java.sql.Statement.executeUpdate(String)">
    <package>java.sql</package>
    <class>Statement</class>
    <method>executeUpdate</method>
    <paramCount>1</paramCount>
    <vulnParam>0</vulnParam>
     <category>SQL Injection</category>
 </sink>
```

Listing 4.2: A Sink Rule from the Security Rules

The security rules are defined on the bases of the literature survey and by studying the documentations available for the methods in Java. The major sources for defining the security rules for JWAST include an already existing tool LAPSE [51], Christian Gotz

[76] and Cheng [77]. The collection of Source and Sink rules used by our tool can be found separately in the Appendix at the end.

4.4.2 Dynamic Analysis Agent Architecture

The dynamic analysis agent is responsible for carrying out the dynamic part of our testing tool. Similar to other analysis agents involved in the testing process, the dynamic analysis agent also takes several inputs that are consumed by different agents acting as sub agents for the dynamic analysis agent.

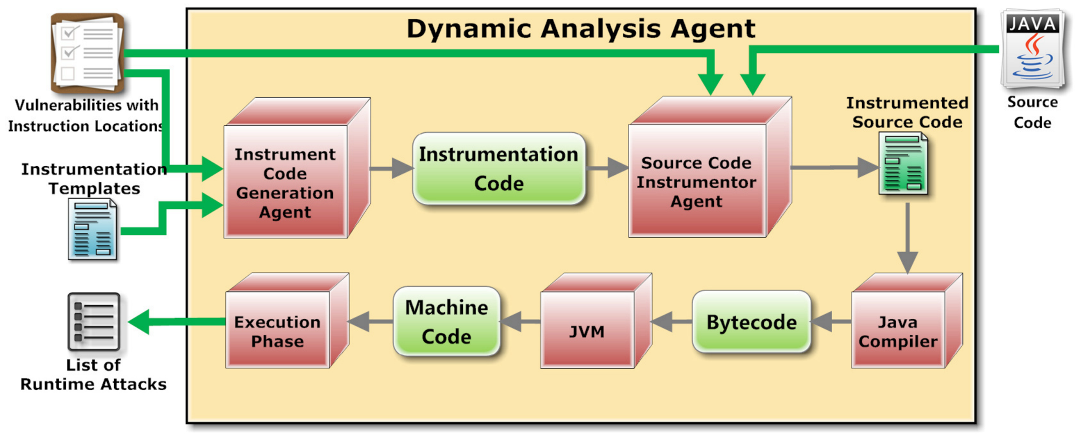

Figure 4.7: Dynamic Analysis Agent Architecture

The architecture diagram of the dynamic analysis agent is shown in Figure 4.7. The operations performed by the dynamic analysis agent are described in the points below:

1. The list of vulnerabilities is given as an input, along with the predefined instrumentation templates, to the instrument code generation agent.
2. The instrument code generation agent generates appropriate instrumentation code based on the vulnerabilities information provided by the vulnerabilities list. This information mainly includes the types of potential vulnerabilities, the

location of vulnerabilities in the source code and the vulnerable method along with the vulnerable parameter of that method.

3. The instrumentation code is passed on to the source code instrumentor agent. This agent also takes the source code of the web application under test and then instruments that code by adding the instrumentation code at appropriate locations.
4. The instrumented source code generated by the source code instrumentor agent is given to the Java compiler which compiles it and produces the bytecode.
5. Bytecode is taken as input by the Java Virtual Machine (JVM), and machine code is produced, which further goes through the execution phase.
6. When the web application which has already been instrumented, runs during the execution, the runtime attacks are detected, prevented and a list of these attacks is generated as a final output of the tool.

Chapter 5

Implementation and Testing

This chapter presents the implementation details followed by the testing that we have performed on our security testing tool called JWAST. The implementation of the tool is presented in details consisting of explanations and discussions of the technologies that we used in implementation. Later in the chapter, various test cases that were run for the tool are presented. The test cases details consist of test inputs, intermediate results produced by the tool and the final outputs in the form of test results

5.1 Implementation

The integrated analysis technique proposed in this thesis and the security testing tool based on the agent architecture presented in Chapter 4 is implemented as a prototype. The implementation of the tool is done in the Java language in the form of independent agent based subsystems as depicted in the architecture of the tool earlier. All the agents of JWAST are written in the Java programming language and for developing, managing and running the agents, JADE framework version 4.3.2 [36] is used as a middleware. The implementation of the tool is done in two phases based on the idea used in the proposed integrated static and dynamic analysis technique. In the first phase, the static analysis is implemented which further consists of subsystems that interact with each other and take the output of one or more subsystems as their input. In the second phase of implementation the dynamic analysis has been implemented based on the results and their format produced during the static analysis. While implementing the static and dynamic analysis modules, a specific mobile agent framework is used. More details about implementation of the complete tool are discussed in the rest of this chapter.

5.1.1 Implementing Static Analysis

For implementing any static code analysis technique, like the one discussed in section 4.2.1, the preprocessing (performing lexical analysis and parsing) is typically required and generally the manual construction of lexers and parsers is rare. Thus a single or a set of tools, depending on the technique under development, are used for automating the lexer and parser construction.

Similarly, before carrying out our static analysis, the source code needs to be preprocessed. Our static analysis technique involves initial phases of a compiler and for this purpose we needed to develop a Java precompiler for lexically analyzing the source code and then parsing it for generating a model. For this purpose there exist multiple tools for Java, known as parser generators. A parser generator can be used to develop your own Java precompiler and then integrate it into your tool for performing required preprocessing the static analysis.

Several parser generators exist and each of them has its own unique set of characteristic features, advantages and disadvantages. We selected Java Compiler Compiler (JavaCC) [78] to be used for generating our parser. Some other parser generator tools include CUP [79], SableCC [80] and ANTLR [81]. Section 5.1.2 explains the process of lexical analyzer and parser generation that we have used for implementing our lexical analyzer and parser using JavaCC.

5.1.1.1 JavaCC

As mentioned earlier, JavaCC is a parser generator that generates lexical analyzer and parser for the Java language. It can only generate a parser in the Java language whereas other tools exist that can generate parsers in different languages. However, we selected JavaCC as our technique detects vulnerabilities in web applications developed in Java only. JavaCC takes an Extended Backus-Naur Form (EBNF) form of language grammar as an input and produces the parser in Java, where the input grammar is not necessarily in Java. The parsers that are generated by the JavaCC are of type LL (k). The generated parser consists of a set of functions and each function has the responsibility to parse individual productions that are given as an input in the EBNF grammar.

5.1.1.2 Lexical Analyzer and Parser Generation

For generating the parser the language grammar with some additional information including Java driver class which is responsible for initiating and starting the generated parser is given to the JavaCC parser generator and it automatically produces the programs capable of analyzing the language described by the input grammar. For implementing our static analysis, we generated our parser using one of the online freely available grammars. We used Java 1.5 grammar [82] which is maintained in an online repository and provided by JavaCC.

5.1.1.3 Building the Abstract Syntax Tree

As we are performing the static analysis on the Abstract Syntax Trees (AST), we need to parse our source code into that model. One of the advantage of using JavaCC to us is that it also generates the AST of the source. However, it does not produce the AST automatically by itself, but we need use a JavaCC based tools called JJTree. So, by the use of JavaCC we create a multiple pass parser and make use of the JJTree, which actually takes a “.jjt” file as an input along with the language grammar file, and generate the AST of the source code under the test.

5.1.1.4 Imports Analysis and Taints Analysis Implementation

Imports analysis and the taints analysis are the heart of static analysis module. Both of the modules perform analysis on the generated abstract syntax tree based on the security rules that are also given as input to these modules. The security rules are written in the XML format which consists of sensitive source and sink methods specifications. The AST generated by the parser is used by imports analyzer and after extracting the imports from the AST, they are analyzed using the defined security rules. If an import matches one of the imports listed in the rules, it is marked as a vulnerable import. Further, for the taints analysis, the AST is visited node by node to find the methods that are marked by the security rules as sensitive sinks. If a sensitive sink method is encountered, more details for that sink are extracted from the security rules. These details mainly include the vulnerable parameter and the vulnerable parameter number. Now, to satisfy the condition for a parameter to be taint, we track the flow of that method parameter from a source method to that identified sink method. The input parameter containing the user input that is taken by the source method is called a taint. Now, if the taint flows from a source method to the sensitive sink method without being validated we mark that statement, containing the sensitive sink method call as a vulnerable statement and then the taints results are return after performing this analysis for the whole AST. The results include the source code file location, line number in the source code, vulnerability type, vulnerable argument and the vulnerable method call.

5.1.2 Implementing Dynamic Analysis

Once the static analysis is completed and the vulnerabilities along with all the details are produced as a result, the dynamic analyzer agent takes control. The implementation of the dynamic analysis is further divided into two modules as in the JWAST architecture presented in section 4.3. The instrument code generation agent implements the traversing and analysis logic. The traversing part implements the logic for visiting the instrumentation templates that are input to the dynamic analysis module. This traversing is performed on the basis of analysis that runs over each vulnerability to identify the vulnerability type and extracts the required parameters like vulnerability location in the source code. Once the vulnerability is identified, the specific instrumentation template is selected and instrumentation method from the source code instrumentor agent is called where it is provided with the instrumentation code and the source code where this is instrumentation code is to be inserted. For instrumenting the source code, the source code processor called Java Instrumentation Engine (JIE) version 1.01 [83] , meant for source code instrumentation, is used. “The Java Instrumentation Engine (JIE) is a generic Java source code processor which inserts instrumentation code at specified locations in a given source code. In its basic mode of operation, JIE receives a Java source file and instrumentation instructions, and emits appropriately transformed Java source code” [83]. Once the source code is instrumented, it is converted to byte code by the Java compiler and then into the machine code by the Java Virtual Machine (JVM) and in the execution phase the instrumentation code gets executed where the runtime attacks are then detected and prevented.

5.1.3 Other Modules

Several modules other than the static and dynamic analysis have been added in the implementation of the tool. This is done to support the tool in performing the security testing. These modules have been added on purpose and they include the modules like configuration, helpers and resources. These are responsible for performing several tasks which may include GUI resources initializations, input source code files filtering, traversing and loading, static analysis rules loading, handling the source code file input streams and other required inter-modules information passing.

One of the important parts in the implementation of static analysis is the XML parser. As the security rules are input to the static analysis in the XML format, a separate Document Object Model (DOM) based XML parser is written in Java which is also the part of implementation.

5.2 Testing the Java Web Application Security Tester

In this section we discuss the importance of testing during and after the development of our tool and we show how the testing is performed. Different testing techniques are applied to ensure the reliability of the developed tool.

Software testing is the process used to help identify the correctness, completeness, security & quality of developed computer software. Finding an error is thus considered a success rather than failure. On finding an error efforts are made to correct it.

For the testing the functionality of our designed tool i.e. JWAST, we have used White-box and Black-box testing techniques. In white box testing, internal code written in every component was tested and it was checked that the code written is efficient in utilizing the resources of the system like memory, band width or the utilization of input/output. To perform Black-box testing on the tool we prepared several formal test case pairs for each type of the vulnerabilities in the class that we have targeted and mentioned in Chapter 2. Each pair in the formal test cases consisted of negative and positive tests where, a negative test is to be performed on the non-vulnerable code and positive test is to be run on the vulnerable code known in advance. The test cases that we have run are designed for different vulnerabilities and are according to the Figure 5.1. The tests that were run are presented in the following sub sections.

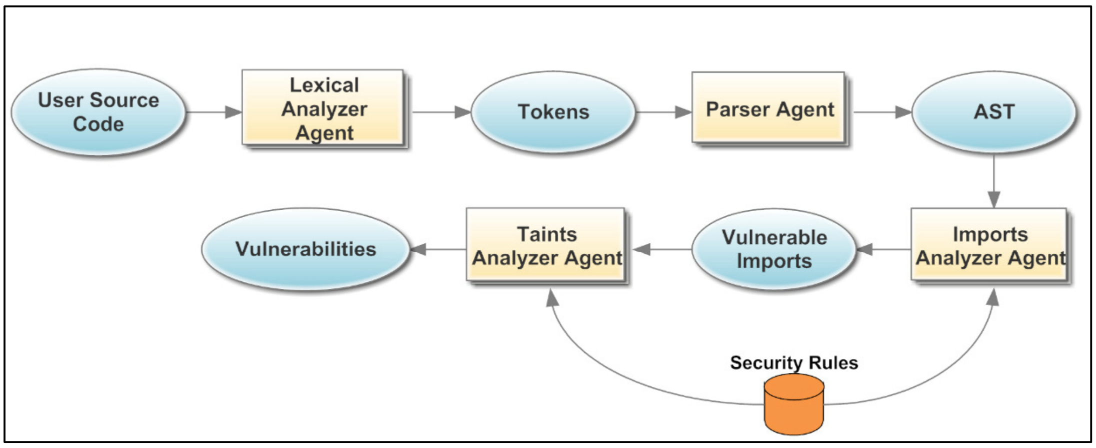

Figure 5.1: JWAST test case design diagram

5.2.1 SQL Injection (Error Path with Vulnerable Code)

This test case is performed to check whether the tool can detect the SQL Injection vulnerability in the given source. The test is presented below where the input is given as vulnerable source code. Then further intermediate states are given as the analysis progresses and until the final test results are produced by the analysis.

5.2.1.1 User Source Code

Figure 5.2 shows the source code that is under test and given to the JWAST as an input. The source code exhibits a known SQL Injection vulnerability in the prepareStatement function call. Whereas, the prepareStatement is a sensitive method as per our rules definition. And this vulnerability can be attacked by providing a carefully crafted input value for the variable “id” that is used as an argument to that sensitive function call.

```
void loginByIdInsecure() throws Exception {
        System.out.println("Half Secure Systems Inc. - login by
id");
        String id = input("User ID?");
        String password = input("Password?");
        try {
            PreparedStatement prep = conn.prepareStatement(
                    "SELECT * FROM USERS WHERE " +
                    "ID=" + id + " AND PASSWORD=?");
            prep.setString(1, password);
            ResultSet rs = prep.executeQuery();
            if (rs.next()) {
                System.out.println("Welcome!");
            } else {
                System.out.println("Access denied!");
            }
            rs.close();
            prep.close();
        } catch (SQLException e) {
            System.out.println(e);
        }
    }
```

Figure 5.2: SQL Injection vulnerable source code

5.2.1.2 Tokens

Upon taking the vulnerable source code the lexical analyzer agent will convert the source code to tokens (Figure 5.3) and pass it to the parser agent where the parser agent will further generate an Abstract Syntax Tree (AST) by parsing the tokens, as shown in Figure 5.1.

```
void, loginByIdInsecure, (, ), throws, Exception, {, System, ., out, ., println, (,
"Half Secure Systems Inc. - login by id", ), ;, String, id, =, input, (, "User ID?",
), ;, String, password, =, input, (, "Password?", ), ;, try, {, PreparedStatement,
prep, =, conn, ., prepareStatement, (, "SELECT * FROM USERS WHERE ",
+, "ID=", +, id, +, " AND PASSWORD=?", ), ;, prep, ., setString, (, 1, ,,
password, ), ;, ResultSet, rs, =, prep, ., executeQuery, (, ), ;, if, (, rs, ., next,
(, ), ), {, System, ., out, ., println, (, "Welcome!", ), ;, }, else, {, System, ., out,
., println, (, "Access denied!", ), ;, }, rs, ., close, (, ), ;, prep, ., close, (, ), ;, },
catch, (, SQLException, e, ), {, System, ., out, ., println, (, e, ), ;, }, },
```

Figure 5.3: The tokenized source code

5.2.1.3 Imports Analysis Results

The AST that is generated by the parser agent is used by the Imports analyzer agent which produces the output as shown in Figure 5.4.

```
java.sql.Connection, java.sql.Statement
```

Figure 5.4: Vulnerable Imports

5.2.1.4 Taints Analysis Results

By using the output produced by the imports analyzer agent and the AST generated by the parser agent, the taints analyzer agent will perform the taint analysis and then produce the final results as the vulnerable statements which are shown in Figure 5.5.

```
D:\Imran KAU\Semester 4\WORKING Folder\Static
Analysis TestCases\SQLInjection.java: Line: 234,
Detected: SQL Injection, Argument: "SELECT * FROM
USERS WHERE " + "ID=" + id + " AND PASSWORD=?",
MethodCall: conn.prepareStatement("SELECT * FROM
USERS WHERE " + "ID=" + id + " AND PASSWORD=?")
```

Figure 5.5: The detected vulnerabilities with their details

5.2.2 SQL Injection (Normal Path with Non-Vulnerable Code)

By running this test case we have checked whether the tool generates the false positives or false negatives. The input, test result and intermediate states are presented below.

5.2.2.1 User Source Code

Figure 5.6 shows the source code under test which is given to the JWAST as an input. The source code executes an SQL query and takes the input “id” from user but do not directly passes it to any sensitive sink method. Therefore, there will not be any security vulnerability detected in this source code.

```
void loginByIdSecure() throws Exception {
        System.out.println("Secure Systems Inc. - login by id");
        String id = input("User ID?");
        String password = input("Password?");
        try {
            PreparedStatement prep = conn.prepareStatement(
                    "SELECT * FROM USERS WHERE " +
                    "ID=? AND PASSWORD=?");
            prep.setInt(1, Integer.parseInt(id));
            prep.setString(2, password);
            ResultSet rs = prep.executeQuery();
            if (rs.next()) {
                System.out.println("Welcome!");
            } else {
                System.out.println("Access denied!");
            }
            rs.close();
            prep.close();
        } catch (Exception e) {
            System.out.println(e);
        }
    }
```

Figure 5.6: The non-vulnerable code without SQL Injection

5.2.2.2 Tokens

The lexical analyzer agent has converted the source code into tokens that are presented in the Figure 5.7.

```
void, loginByIdSecure, (, ), throws, Exception, {,
System, ., out, ., println, (, "Secure Systems Inc. -
login by id", ), ;, String, id, =, input, (, "User
ID?", ), ;, String, password, =, input, (,
"Password?", ), ;, try, {, PreparedStatement, prep,
=, conn, ., prepareStatement, (, "SELECT * FROM USERS
WHERE ", +, "ID=? AND PASSWORD=?", ), ;, prep, .,
setInt, (, 1, ,, Integer, ., parseInt, (, id, ), ),
;, prep, ., setString, (, 2, ,, password, ), ;,
ResultSet, rs, =, prep, ., executeQuery, (, ), ;, if,
(, rs, ., next, (, ), ), {, System, ., out, .,
println, (, "Welcome!", ), ;, }, else, {, System, .,
out, ., println, (, "Access denied!", ), ;, }, rs, .,
close, (, ), ;, prep, ., close, (, ), ;, }, catch, (,
Exception, e, ), {, System, ., out, ., println, (, e,
), ;, }, },
```

Figure 5.7: The tokenized source code

5.2.2.3 Imports Analysis Results

As the input source code is trying to execute an SQL query, vulnerable import statements will be presented and are detected by the imports analysis as presented in the Figure 5.8

```
java.sql.Connection, java.sql.Statement
```

Figure 5.8: Vulnerable Imports

5.2.2.4 Taints Analysis Results

The taints analysis is performed by the taints analyzer agent, after the vulnerable imports analysis and the final test results are presented in the Figure 5.9.

```
The Static Analysis has detected no Vulnerabilities.
```

Figure 5.9: Final results generated by the analysis

5.2.3Cross-Site Scripting (Error Path with Vulnerable Code)

By running this test case we have checked whether the tool generates the false positives or false negatives in case of the Cross-site Scripting vulnerabilities. The input, test result and intermediate states are presented below.

5.2.3.1 User Source Code

Figure 5.10 shows the source code under test which is given to the JWAST as an input. In this source code the XSS vulnerability exists as the input is being taken from the request header and being passed to sensitive sink method of the PrintWriter class. So, as the input is used unvalidated and exactly in the format in which the user inputted it, there can be an XSS attack if the malicious input is provided.

```
import java.io.PrintWriter;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.*;

public class HelloServlet extends HttpServlet {


public void doGet (HttpServletRequest req,
                       HttpServletResponse res)
        throws ServletException, IOException
        {
                String input = req.getHeader("USERINPUT");

                PrintWriter out = res.getWriter();
                out.println(input);
                out.close();
        }
}
```

Figure 5.10: The vulnerable code with XSS Vulnerability

5.2.3.2 Tokens

The lexical analyzer agent has converted the source code into tokens that are presented in the Figure 5.11.

```
import, java, ., io, ., *, ;, import, javax, ., servlet, .,
http, ., *, ;, import, javax, ., servlet, ., *, ;, public,
class, HelloServlet, extends, HttpServlet, {, public, void,
doGet, (, HttpServletRequest, req, ,, HttpServletResponse, res,
), throws, ServletException, ,, IOException, {, String, input,
=, req, ., getHeader, (, "USERINPUT", ), ;, PrintWriter, out, =,
res, ., getWriter, (, ), ;, out, ., println, (, input, ), ;,
out, ., close, (, ), ;, }, },
```

Figure 5.11: The tokenized Source Code

5.2.3.3 Imports Analysis Results

The imports analyzer agent detected the vulnerable imports in the source code. As we know, the code is vulnerable is vulnerable to the XSS attacks, the imports analyzer agent returned the import statements that are XSS vulnerable. The results for the imports analysis are shown in Figure 5.12.

```
Vulnerable Import: java.io.PrintWriter
```

Figure 5.12: Vulnerable Imports

5.2.3.4 Taints Analysis Results

The taints analysis is performed by the taints analyzer agent, after the vulnerable imports analysis and the final test results are presented in the Figure 5.13.

```
The Static Analysis has detected the Following Vulnerabilities:

D:\Imran KAU\Semester 4\WORKING Folder\Static Analysis TestCases\New
folder\XSS.java: Line: 13, Detected: Cross-site Scripting, Argument:
input, MethodCall: out.println(input)
```

Figure 5.13: Final results generated by the analysis

5.2.4 Cross-Site Scripting (Normal Path with Non-Vulnerable Code)

After testing the correctness of our tool for the XSS vulnerable code, we also tested it with the non XSS vulnerable code. The results showed that our tool did not report any false positive as we run this test case. The intermediate and final results are shown in the following sub sections.

5.2.4.1 User Source Code

Figure 5.14 shows the source code which performs the validation and encodes the input for removing any possible html characters or scripts from the user input before passing it to the XSS sensitive sink method. As a result, the XSS vulnerability is removed from this code and the XSS attacks would not be possible. We have given this source code as an input to our tool, and our tool did not report any vulnerabilities, which proves the correctness of our tool.

```
import java.io.PrintWriter;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.*;

public class HelloServlet extends HttpServlet {

        public void doGet (HttpServletRequest req,
                               HttpServletResponse res)
                throws ServletException, IOException
        {
                String input = req.getHeader("USERINPUT");

                PrintWriter out = res.getWriter();
                String encodedInput = TextUtils.htmlEncode(input);
                out.println(encodedInput);
                out.close();
        }
}
```

Figure 5.14: The non-vulnerable code with XSS Vulnerability

5.2.4.2 Tokens

The lexical analyzer agent has converted the source code into tokens that are presented in the Figure 5.15.

```
import, java, ., io, ., PrintWriter, ;, import, javax, .,
servlet, ., http, ., HttpServletRequest, ;, import, javax, .,
servlet, ., *, ;, public, class, HelloServlet, extends,
HttpServlet, {, public, void, doGet, (, HttpServletRequest, req,
,, HttpServletResponse, res, ), throws, ServletException, ,,
IOException, {, String, input, =, req, ., getHeader, (,
"USERINPUT", ), ;, PrintWriter, out, =, res, ., getWriter, (, ),
;, String, encodedInput, =, TextUtils, ., htmlEncode, (, input,
), ;, out, ., println, (, encodedInput, ), ;, out, ., close, (,
), ;, }, },
```

Figure 5.15: The tokenized Source Code

5.2.4.3 Imports Analysis Results

In this case the imports analyzer detected the vulnerable import as similar to the vulnerable code. But this does not mean that there is a vulnerability in the source code. The final decision about the presence of vulnerability will be made by the taints analyzer agent. The vulnerable imports reported by the tool are shown in Figure 5.16.

```
Vulnerable Import: java.io.PrintWriter
```

Figure 5.16: Vulnerable Imports

5.2.4.4 Taints Analysis Results

The taints analysis is performed by the taints analyzer agent, and as there was no unvalidated input being used in the sensitive sink method, the tool does not report any vulnerability. The result is shown in the Figure 5.17.

```
The Static Analysis has detected no Vulnerabilities.
```

Figure 5.17: Final results generated by the analysis

Chapter 6

Evaluation and Comparative Study

This chapter describes the assessment criteria used for evaluation as we evaluated our tool for its performance and features that it provides. For the evaluation we have used specific metrics that are also discussed in this chapter. Also, in this chapter we have showed the comparison of our tool with some of the other tools targeting the security problem.

6.1 Evaluation Metrics

For evaluating the performance of our tool, we have used some performance metrics which makes use of the data based on total number of True Positives (TP), True Negatives (TN), False Positives (FP) and False Negatives (FN).

In our case these terms can be further defined as:

True Positive

The source code has a vulnerability and the tool reports that vulnerability.

True Negative

The source code does not have a vulnerability and the tool does not report that vulnerability.

.False Positive

The source code does not have a vulnerability and the tool reports that vulnerability.

False Negative

The source code has a vulnerability and the tool does not report that vulnerability.

For evaluation of our tool we calculated true positives, true negatives, false positives and false negatives. Later with the help of these, we calculated the metrics including precision, accuracy and recall. For calculating these metrics the values gained from the true positives, false positives, true negatives and false negatives are used in a defined formula made for calculating the precision, accuracy and recall. The precision describes how well a tool does identify bugs whereas, the accuracy shows the ability of tool to differentiate between vulnerable and non-vulnerable code and the recall value tells the soundness or sensitivity of the tool. The maximum value of the recall can be 1 and the minimum value can be 0. Where, if the recall is 1, it means the tool can detect all the vulnerabilities [18].

For calculating the metrics described above, following formula [84] is used:

$$Precision = \frac{True\ Positives}{True\ Positives + False\ Positives}$$

$$Accuracy = \frac{True\ Positives + True\ Negatives}{True\ Positives + True\ Negatives + False\ Positives + False\ Negatives}$$

$$Recall = \frac{True\ Positives}{True\ Positives + False\ Negatives}$$

6.2 Evaluation Results

For evaluating the JWAST, we used tests suites provided by Software Assurance Metrics And Tool Evaluation (SAMATE) [85]. They provide several test suites for performing security tools evaluation and we have used the Juliet Test Suite for Java version 1.1.1 and Juliet Test Suite for Java version 1.2. Table 6.1 shows the summary of the results that we have obtained by running JWAST on the Juliet Test Suite version 1.1.1 and the Table 6.2 shows the results that are obtained by the Juliet Test Suite version 1.2.

Juliet Test Case	**Total Test Cases**			**Precision**	**Accuracy**	**Recall**
Version 1.1.1	6330	True Positives	1250	**0.13**	**0.6**	**0.2**
		False Positives	8421			
		True Negatives	18892			
		False Negatives	5062			
Version 1.2	9731	True Positives	2008	**0.4**	**0.8**	**0.2**
		False Positives	3180			
		True Negatives	40932			
		False Negatives	7723			

Table 6.1: Results of test run on Juliet Test Case version 1.1.1 and Version 1.2

6.3 Performance Evaluation

As we have implemented our tool as an agent based testing tool, we have gained several advantages inherent in the mobile agents. We have evaluated our agent based security testing tool JWAST, on the basis of following points.

6.3.1 Platform Independence

As our tool is an agent based tool, it is completely platform independent and can be run on any platform by sending the agent instances to perform testing.

6.3.2 Remote Testing

Unlike the tools that are implementing using conventional desktop based technologies, we can send instances of JWAST agent to the remote site of target web application for remote testing.

6.3.3 Concurrent Testing

Our tool enables the users to perform concurrent testing. This can be achieved by creating multiple instances of JWAST agent and use each instance to test different web application at the same time.

6.3.4 Increased Throughput

As the JWAST provides concurrency in the testing, the total throughput can be increased by testing more number of web application in a specific time period.

6.3.5 Reduced Network Traffic

The use of JWAST will help in reducing the network load. This can be achieved by sending a mobile agent instance of the JWAST to remote site location where web application source code is deployed, instead of downloading the source code for testing on the local machine.

6.3.6 High Scalability

JWAST is highly scalable in nature and it can be scaled up to test any number of web applications. This is due to the fact that any number of instances of JWAST can be created and used for testing the web applications.

6.4 Java Web Application Security Tester Coverage Comparison

As there exist several tools that are meant for security testing, presented in the Chapter 3. Here, we present a coverage comparison of our tool with some of the other tools built for the web application security testing.

Vulnerabilities/Tools	JWAST	PMD	Find Bugs	RIPS	SAFELI
SQL injection	✓	✓	✓	✓	✓
Cross-Site Scripting	✓	✓	✓	✓	×
Http Response Splitting	✓	×	×	✓	×
Path Traversal	✓	×	×	×	×
Command Injection	✓	×	✓	✓	×
XPath Injection	✓	×	×	×	×
LDAP Injection	✓	×	×	×	×

Table 6.2: Coverage comparison of JWAST with other tools

Table 6.3 shows the difference in the vulnerability coverage that exists between our tool and other tools. As it shown in the table, our tool JWAST covers all the listed vulnerabilities. JWAST is capable of detecting and preventing the SQL injection, XSS, Http response splitting, path traversal, command injection, XPath injection and LDAP injection vulnerabilities and runtime attacks. However, the other tools that we have compared with, cover only few of these vulnerabilities where path traversal, XPath injection and LDAP injection vulnerabilities and attacks can only be detected using our tool, JWAST.

6.5 Java Web Application Security Tester Feature Comparison

This sections presents the comparative study that we have made for our tool with the other existing tools. In the Table 6.4 we have shown the differences in the implemented features of our tool with other tools. This comparison shows that our tool has contributed in terms of several improvements in the features that current tools offer.

Features/Tools	**JWAST**	**PMD**	**Find Bugs**	**RIPS**	**Dytan**
Underlying Technology	Agent Based	Conventional Desktop Based	Conventional Desktop Based	Conventional Desktop Based	Desktop Based
Techniques Used	Static + Dynamic Analysis	Static Analysis	Static Analysis	Static Analysis	Dynamic Analysis
Target Language	JAVA	JAVA	JAVA	PhP	binaries
Input Format	Source Code	Source Code	Byte Code	Source Code	x86 binaries
Extensibility	Yes	Yes	Yes	No	Yes

Table 6.3: Feature comparison of JWAST with other tools

As it can be seen in the Table 6.4, our tool JWAST differs in the underlying technology, where we have introduced an agent based security testing tool as compared to the other existing tools which developed as a conventional desktop based tools. Also, the technique that we have introduced and used in our tool is an integrated static and dynamic analysis based technique which is in contrast to the other tools that use either only static or only dynamic analysis techniques for the testing. One other obvious advantage of our tool over the other tools is the input format used for performing the security testing, which is the source code instead of the binaries or the bytecode. This enable the users to perform the security testing even if the application is under development and the bytecode or binaries are not available yet.

Chapter 7

Conclusion and Future Work

This chapter highlights the main points and concludes the work that has been done in this thesis and the achievements that have been made are discussed. Furthermore, the thoughts and suggestions of potential future work and improvements, which could not be included, are also presented in this chapter.

7.1 Conclusion

The demand of web applications has been boosted up in the last decade as more and more services are shifted on the World Wide Web. Whereas, the web applications are highly exposed and vulnerable in nature and in addition these are poorly programmed with respect to the security. Various testing techniques and tools for software are well developed and are in place. However, the software security problem is neglected by not performing the security testing of the applications. By the passage of time, the number of vulnerabilities and attacks are increasing in the web based applications and among them the input validation vulnerabilities are the most common problem. One of the key security measures that needs to be taken in order to mitigate the increasing number of vulnerabilities in web applications, is to automate the security testing process as much as possible.

In this research, we have introduced an optimized taints based static analysis technique that has developed keeping in view the ability for this technique to be integrated with a dynamic analysis technique. A set of security rules has been defined to support our static taints analysis. These rules enable the static analysis to identify the potential vulnerability sources in the given web application source code. A dynamic analysis technique has also been developed and optimized to work in combination with the static analysis technique. The dynamic analysis technique plays the role of double checker in detection and also helps in the prevention of web application attacks which might be possible due to the present vulnerabilities. A set of instrumentation templates has been introduced to be used with the dynamic analysis empower, the web application under test, to stop the web attacks from happening. On the basis of our integrated static and dynamic analysis technique we built a web application security testing tool called JWAST. JWAST enables the programmers and software testers to perform security testing on web based applications written in Java.

For proving our technique as reliable and efficient, we have implemented our tool in Java as an agent based security testing tool for testing the web based applications written in Java. We have conducted several experiments to test the ability of JWAST to detect input validation vulnerabilities. We designed and run several positive and negative test cases for each type of vulnerability in the input validation vulnerabilities class. The results have shown that our tool detects and prevents the input validation vulnerabilities and is sound with respect to its rule base.

We also compared our tool with other existing tools and the comparative study revealed that our tool performs better than the other tools we compared to. Our tool provides improved coverage in terms of support for number and types of security vulnerabilities as compared to the other tools. Also, the features that are provided by our tool are better than the other tools we compared to. Namely, the underlying technology, the integrated testing technique and the input format as a source code are the features where our tool takes an edge over other tools.

7.2 Future Work

Although we have shown that our integrated static and dynamic analysis technique enable our security testing tool more efficient than other closely related works, but future evaluations of the JWAST should be focused on detection and prevention of more vulnerabilities in a more efficient way and for other languages also.

7.2.1 Performance Enhancements

For future work, we will investigate further the applicability of integrating such a technique that can reduce the source code size. This will help our security testing tool to compete the testing and generate results in less time. One of the possibility for achieving this goal can be to integrate code slicing in the current technique. Code slicing is a technique that is used to eliminate the code that is never used. Performing code slicing will result in the code reduction and our overall testing technique will be improved.

The security rules used by the static analyzer agent can be enhanced. We can include the support of cleanse rules in our technique. Cleanse rules are the rules that explicitly specify the user defined validation functions that can be called as cleanse methods. While performing the static taint analysis, the cleanse rule can help our analysis know about the user defined validation functions and in this way the static analysis will not report a vulnerability and the taint will be removed, if in case a cleanse method is encountered.

7.2.2 More Vulnerabilities Classes Support

Current static taint analysis technique is based on data flow analysis. Using this technique we are able to detect taint style vulnerabilities which are also known as input validation vulnerabilities. However, as a future work we will extend our tool coverage and support the detection of more vulnerabilities of input validation class and other classes as well. To achieve this goal we will integrate more advanced static analysis technique into our tool. Some of the advanced static analysis techniques which can be used include symbolic analysis and abstract interpretation [86].

7.2.3 Other Languages Support

Current form of the JWAST can only be used to test the web based applications developed in Java. However, in the future we will be extending the implementation of our technique to other languages which will enable our tool to test the applications written in languages other than Java. The current proposal for the extension of static analysis implementation is to generate parsers for other languages and perform taint analysis. And for the extension of dynamic analysis, the language specific instrumentor implementation, for example for .NET by the use of CLR Profiler [87], will be provided so that the application source code written in other languages can also be instrumented for the dynamic analysis.

LIST OF REFERENCES

[1] L. Shklar and R. Rosen, *Web application architecture: Principles, protocols and practices*. John Wiley & Sons, 2004.

[2] N. Teodoro and C. Serrao, "Web application security: Improving critical web-based applications quality through in-depth security analysis," in *Information Society (i-Society), 2011 International Conference*, 2011, pp. 457–462.

[3] Cenzic, "Application Vulnerability Trends Report:2013," Dec. 2013.

[4] I. S. Systems, "IBM X-Force Threat Intelligence Quarterly 2014," 2014.

[5] H. Herbert and others, "Why security testing is hard," *IEEE Secur. Priv.*, 2003.

[6] G. a. Di Lucca and A. R. Fasolino, "Testing Web-based applications: The state of the art and future trends," *Inf. Softw. Technol.*, vol. 48, no. 12, pp. 1172–1186, Dec. 2006.

[7] G. of the HKSAR, "WEB ATTACKS AND COUNTERMEASURES," 2008.

[8] R. Hower, "Software QA and Testing Resource Center." [Online]. Available: http://www.softwareqatest.com/. [Accessed: 15-Nov-2013].

[9] A. Tappenden, P. Beatty, J. Miller, A. Geras, and M. Smith, "Agile security testing of web-based systems via httpunit," in *Agile Conference, 2005. Proceedings*, 2005, pp. 29–38.

[10] B. Arkin, S. Stender, and G. McGraw, "Software penetration testing," *IEEE Secur. Priv.*, vol. 3, no. 1, pp. 84–87, 2005.

[11] C. Kruegel, G. Vigna, and W. Robertson, "A multi-model approach to the detection of web-based attacks," *Comput. Networks*, vol. 48, no. 5, pp. 717–738, 2005.

[12] C. Kruegel and G. Vigna, "Anomaly detection of web-based attacks," in *Proceedings of the 10th ACM conference on Computer and communications security*, 2003, pp. 251–261.

[13] M. Johns and J. Winter, "RequestRodeo: Client side protection against session riding," in *Proceedings of the OWASP Europe 2006 Conference*, 2006.

[14] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic, "Noxes: a client-side solution for mitigating cross-site scripting attacks," in *Proceedings of the 2006 ACM symposium on Applied computing*, 2006, pp. 330–337.

[15] Paros and Yukusan, “Paros.” [Online]. Available: http://www.parosproxy.org/index.shtml. [Accessed: 15-Nov-2013].

[16] R. S. Cox, J. G. Hansen, S. D. Gribble, and H. M. Levy, “A safety-oriented platform for web applications,” in *Security and Privacy, 2006 IEEE Symposium on*, 2006, p. 15–pp.

[17] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, “Terra: A virtual machine-based platform for trusted computing,” in *ACM SIGOPS Operating Systems Review*, 2003, vol. 37, no. 5, pp. 193–206.

[18] J. Shrestha, “Static Program Analysis,” 2013.

[19] G. McGraw, “Static Analysis for Security, Building Security In,” *IEEE Secur. Priv.*, vol. 2, no. 6, pp. 76–79, 2000.

[20] A. Hicken, “.NET Static Analysis and Parasoft dotTEST.” 2011.

[21] OWASP, “OWASP Top 10 - 2013 The Ten Most Critical Web Application Security Risks,” 2013.

[22] “Open Web Application Security Project (OWASP).” [Online]. Available: https://www.owasp.org/index.php/Main_Page. [Accessed: 10-Oct-2013].

[23] Imperva, “White Paper Imperva ’ s Web Application Attack Report Ed.3,” 2012.

[24] J. Tudor, “Web Application Vulnerability Statistics 2013,” 2013.

[25] M. Dowd, J. McDonald, and J. Schuh, *The Art of Software Security Assessment - Identifying and Preventing Software Vulnerabilities*. Addison Wesley Professional, 2006, p. 1200.

[26] C. E. Landwehr, A. R. Bull, J. P. McDermott, and W. S. Choi, “A taxonomy of computer program security flaws,” *ACM Comput. Surv.*, vol. 26, no. 3, pp. 211–254, 1994.

[27] M. Howard, D. LeBlanc, and J. Viega, *19 deadly sins of software security*. McGraw-Hill/Osborne California, 2005.

[28] M. Howard, D. LeBlanc, and J. Viega, “24 Deadly Sins of Software Security,” *Sin*, vol. 11, p. 183, 2009.

[29] K. Tsipenyuk, B. Chess, and G. McGraw, “Seven Pernicious Kingdoms : A Taxonomy of Software Security Errors,” *Secur. Privacy, IEEE*, pp. 81–85, 2005.

[30] B. Martin, M. Brown, A. Paller, D. Kirby, and S. Christey, “2011 CWE/SANS Top 25 Most Dangerous Software Errors,” *Common Weakness Enumer.*, vol. 7515, 2011.

[31] OWASP, “Web Application Attacks.” [Online]. Available: https://www.owasp.org/index.php/Attacks. [Accessed: 01-Oct-2013].

[32] R. Gray, D. Kotz, S. Nog, D. Rus, and G. Cybenko, “Mobile agents: the next generation in distributed computing,” *Proc. IEEE Int. Symp. Parallel Algorithms Archit. Synth.*, pp. 8–24, 1997.

[33] P. Braun and W. R. Rossak, *Mobile agents: Basic concepts, mobility models, and the tracy toolkit*. Elsevier, 2005.

[34] Y.-W. Huang, C.-H. Tsai, T.-P. Lin, S.-K. Huang, D. T. Lee, and S.-Y. Kuo, “A testing framework for Web application security assessment,” *Comput. Networks*, vol. 48, no. 5, pp. 739–761, Aug. 2005.

[35] S. Paydar, “An Agent-Based Framework for Automated Testing of Web-Based Systems,” *J. Softw. Eng. Appl.*, vol. 04, no. 02, pp. 86–94, 2011.

[36] T. I. S.p.A., “JADE.” [Online]. Available: http://jade.tilab.com/. [Accessed: 10-Jan-2014].

[37] J. Viega, J.-T. Bloch, Y. Kohno, and G. McGraw, “ITS4: A static vulnerability scanner for C and C++ code,” in *16th Annual Computer Security Applications Conference*, 2000.

[38] D. A. Wheeler, “Flawfinder.” [Online]. Available: http://www.dwheeler.com/flawfinder/. [Accessed: 02-Jan-2014].

[39] I. Secure Software, “Rough auditing tool for security (RATS),” 2001. [Online]. Available: https://www.fortify.com/ssa-elements/threat-intelligence/rats.html. [Accessed: 15-Dec-2013].

[40] D. Evans, J. Guttag, J. Horning, and Y. M. Tan, “LCLint: A tool for using specifications to check code,” in *2nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, 1994.

[41] D. Larochelle and D. Evans, “Statically detecting likely bu er overflow vulnerabilities,” in *10th USENIX Security Symposium*, 2001.

[42] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken, “A first step towards automated detection of bu er overrun vulnerabilities,” in *7th Networking and Distributed System Security Symposium*, 2000.

[43] Y. Xie, A. Chou, and D. Engler, “ARCHER: Using symbolic, path- sensitive analysis to detect memory access errors,” in *n Proceedings of the 9th European Software Engineering Conference 10th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2003.

[44] J. S. Foster, M. Fähndrich, and A. Aiken, “A Theory of Type Qualifiers,” in *ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, 1999.

[45] G. Holzmann, "Static Source Code Checking for User-defined Properties," in *Pasadena, CA, USA*, 2002.

[46] A. I. SOTIROV, "AUTOMATIC VULNERABILITY DETECTION USING STATIC SOURCE CODE ANALYSIS," 2005.

[47] H. Chen and D. Wagner, "MOPS: an infrastructure for examining security properties of software," in *9th ACM Conference on Computer and Communications Security*, 2002.

[48] V. B. Livshits and M. S. Lam, "Finding Security Vulnerabilities in Java Applications with Static Analysis," in *Usenix Security Symposium*, 2005.

[49] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy : A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper)," 2006.

[50] Y. Xie and A. Aiken, "Static Detection of Security Vulnerabilities in Scripting Languages," in *Usenix Security Symposium*, 2006.

[51] B. Livshits, "Griffin Software Security Project." .

[52] "Checkstyle 5.7." [Online]. Available: http://checkstyle.sourceforge.net/. [Accessed: 01-Dec-2013].

[53] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended static checking for Java," *ACM SIGPLAN Not.*, vol. 37, no. 5, p. 234, May 2002.

[54] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *ACM SIGPLAN Not.*, vol. 39, no. 12, p. 92, Dec. 2004.

[55] X. Fu, X. Lu, B. Peltsverger, S. Chen, K. Qian, and L. Tao, "A Static Analysis Framework For Detecting SQL Injection Vulnerabilities," *31st Annu. Int. Comput. Softw. Appl. Conf. - Vol. 1- (COMPSAC 2007)*, pp. 87–96, Jul. 2007.

[56] S. Thomas, L. Williams, and T. Xie, "On automated prepared statement generation to remove SQL injection vulnerabilities," *Inf. Softw. Technol.*, vol. 51, no. 3, pp. 589–598, Mar. 2009.

[57] "PMD." [Online]. Available: http://pmd.sourceforge.net/. [Accessed: 10-Jan-2014].

[58] M. Aderhold, "Tailoring PMD to Secure Coding," 2013.

[59] V. Haldar, D. Chandra, and M. Franz, "Dynamic taint propagation for Java," in *Computer Security Applications Conference, 21st Annual*, 2005, p. 9–pp.

[60] G. T. Buehrer, B. W. Weide, and P. A. G. Sivilotti, "Using Parse Tree Validation to Prevent SQL Injection Attacks," in *International Workshop on Software Engineering and Middleware (SEM) at Joint FSE and ESEC*, 2005.

[61] S. Bandhakavi, P. Bisht, P. Madhusudan, and V. N. Venkatakrishnan, “CANDID: preventing sql injection attacks using dynamic candidate evaluations,” in *Proceedings of the 14th ACM conference on Computer and communications security*, 2007, pp. 12–24.

[62] S. Boyd and A. D. Keromytis, “SQLrand: preventing SQL injection attacks,” in *Applied Cryptog- raphy and Network Security Conference*, 2004.

[63] W. Chang, B. Streiff, and C. Lin, “Efficient and extensible security enforcement using dynamic data flow analysis,” in *Proceedings of the 15th ACM conference on Computer and communications security*, 2008, pp. 39–50.

[64] W. Xu, S. Bhatkar, and R. Sekar, “Practical Dynamic Taint Analysis for Countering Input Validation Attacks on Web Applications,” pp. 1–15.

[65] J. Newsome and D. Song, “Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software,” 2005.

[66] D. Scott and R. Sharp, “Abstracting application-level web security,” in *Proceedings of the 11th international conference on World Wide Web*, 2002, pp. 396–407.

[67] A. N. Salvatore, G. Doug, and G. David, “Automatically Hardening Web Applications Using Precise Tainting,” no. December, 2004.

[68] William G. J. Halfond Alessandro Orso and P. Manolios, “Using Positive Tainting and Syntax-Aware Evaluation to Counter SQL Injection Attacks,” in *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE 2006)*, 2006.

[69] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, “Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis,” 2007.

[70] G. A. Di Lucca, A. R. Fasolino, M.Mastroianni, and P.Tramontana, “Identifying Cross Site Scripting Vulnerabilities in Web Applications,” 2004.

[71] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, and C. K. G. Vigna, “Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications,” *IEEE Secur. Priv.*, 2008.

[72] Halfond, W. GJ, and A. Orso, “AMNESIA : Analysis and Monitoring for NEutralizing SQL-Injection Attacks,” in *20th IEEE/ACM international Conference on Automated software engineering*, 2005.

[73] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo, “Securing web application code by static analysis and runtime protection,” *Proc. 13th Conf. World Wide Web - WWW ’04*, p. 40, 2004.

[74] U. Erlingsson and F. Schneider, “SASI enforcement of security policies: a retrospective,” in *New Security Paradigms Workshop*, 2000.

[75] A. C. Myers, “JFlow: Practical Mostly-Static Information Fow Control,” in *Symposium on Principles of Programming Languages*, 1999.

[76] C. Gotz, “Vulnerability identification in web applications through static analysis,” 2013.

[77] G. L. Cheng, “Java Security Vulnerabilities Detection with Static Analysis,” 2007.

[78] S. Microsystems, “JavaCC.” [Online]. Available: https://javacc.java.net/. [Accessed: 17-Nov-2013].

[79] S. E. Hudson, “CUP User’s Manual.” Usability Center, Georgia Institute of Technology, 1999.

[80] E. M. Gagnon and L. J. Hendren, “SableCC, an object-oriented compiler framework,” in *Technology of Object-Oriented Languages, 1998. TOOLS 26. Proceedings*, 1998, pp. 140–154.

[81] T. J. Parr, “Language translation using PCCTS and C++(a reference guide),” *Parr Res. Corp.*, 1995.

[82] “JavaCC Grammars Repository.” [Online]. Available: https://java.net/projects/javacc/downloads/directory/contrib/grammars. [Accessed: 15-Dec-2013].

[83] E. Tromer, “Java Instrumentation Engine.” [Online]. Available: http://cs.tau.ac.il/~tromer/jie/. [Accessed: 05-Feb-2014].

[84] Wikipedia, “Precison, Accuracy and Recall.” [Online]. Available: http://en.wikipedia.org/wiki/Precision_and_recall. [Accessed: 01-Mar-2014].

[85] NIST, “SAMATE.” [Online]. Available: http://samate.nist.gov/Main_Page.html. [Accessed: 07-Feb-2014].

[86] W. Wolfgang, “A Survey of Static Program Analysis Techniques,” *Vienna Univ. Technol.*, pp. 1–16, 2005.

[87] Microsoft, “CLR Profiler for .NET Framework 4.” [Online]. Available: http://www.microsoft.com/en-us/download/details.aspx?id=16273. [Accessed: 10-Mar-2014].

APPENDIX: A
Source Rules and Sink Rules

Source Rules

```xml
<?xml version="1.0" encoding="UTF-8"?>

<sources>
    <!-- Parameter Tampering-->
    <source id="javax.servlet.ServletRequest.getParameter(String)">
        <package>javax.servlet</package>
        <class>ServletRequest</class>
        <method>getParameter</method>
        <category>Untrusted Source</category>
    </source>

    <source id="javax.servlet.ServletRequest.getParameterValues(String)">
        <category>Parameter Tampering</category>
    </source>

    <source id="javax.servlet.ServletRequest.getParameterMap()">
        <category>Parameter Tampering</category>
    </source>

    <source id="javax.servlet.ServletRequest.getParameterNames()">
        <category>Parameter Tampering</category>
    </source>

    <source id="javax.servlet.http.HttpServletRequest.getRequestedSes-
sionId()">
        <category>Parameter Tampering</category>
    </source>

    <source id="javax.servlet.http.HttpServletRequest.getQueryString()">
        <category>Parameter Tampering</category>
    </source>

    <source id="javax.servlet.http.HttpServletRequest.getRemoteUser()">
        <category>Parameter Tampering</category>
    </source>

    <source id="javax.servlet.http.HttpServletRequest.getCookies()">
        <category>Parameter Tampering</category>
    </source>

    <!-- Header Manipulation -->
    <source id="javax.servlet.ServletRequest.getScheme()">
        <category>Header Manipulation</category>
    </source>

    <source id="javax.servlet.ServletRequest.getProtocol()">
        <category>Header Manipulation</category>
    </source>

    <source id="javax.servlet.ServletRequest.getContentType()">
        <category>Header Manipulation</category>
    </source>

    <source id="javax.servlet.ServletRequest.getServerName()">
        <category>Header Manipulation</category>
    </source>
    <source id="javax.servlet.ServletRequest.getRemoteAddr()">
        <category>Header Manipulation</category>
    </source>

    <source id="javax.servlet.ServletRequest.getRemoteHost()">
        <category>Header Manipulation</category>
    </source>

    <source id="javax.servlet.ServletRequest.getRealPath()">
        <category>Header Manipulation</category>
    </source>

    <source id="javax.servlet.ServletRequest.getLocalName()">
        <category>Header Manipulation</category>
    </source>

    <source id="javax.servlet.ServletRequest.getLocalAddr()">
        <category>Header Manipulation</category>
    </source>

    <source id="javax.servlet.ServletRequest.getAuthType()">
        <category>Header Manipulation</category>
    </source>

    <source id="javax.servlet.ServletRequest.getRequestURI()">
        <category>Header Manipulation</category>
    </source>

    <source id="javax.servlet.ServletRequest.getRequestDis-
patcher(String)">
        <category>Header Manipulation</category>
    </source>

    <source id="javax.servlet.http.HttpServletRequest.getMethod()">
        <category>Header Manipulation</category>
    </source>

    <source id="javax.servlet.http.HttpServletRequest.getContentType()">
        <category>Header Manipulation</category>
    </source>

    <source id="javax.servlet.http.HttpServletRequest.getCon-
tentLength()">
        <category>Header Manipulation</category>
    </source>

    <source id="javax.servlet.http.HttpServletRequest.getRemoteUser()">
        <category>Header Manipulation</category>
    </source>

    <source id="javax.servlet.http.HttpServletRequest.getQueryString()">
        <category>Header Manipulation</category>
    </source>

    <source id="javax.servlet.http.HttpServletRequest.getPathInfo()">
        <category>Header Manipulation</category>
    </source>
    <source id="javax.servlet.http.HttpServletRequest.getHeader()">
        <category>Header Manipulation</category>
    </source>
```

```xml
<source id="javax.servlet.http.HttpServletRequest.getHeaders()">
    <category>Header Manipulation</category>
</source>

<!-- URL Tampering -->
<source id="javax.servlet.http.HttpServletRequest.getRequestURL()">
    <category>URL Tampering</category>
</source>

<source id="javax.servlet.http.HttpServletRequest.getRequestURI()">
    <category>URL Tampering</category>
</source>

<!-- Cookie Poisoning -->
<source id="javax.servlet.http.Cookie.getValue()">
    <category>Cookie Poisoning</category>
</source>

<source id="javax.servlet.http.Cookie.getPath()">
    <category>Cookie Poisoning</category>
</source>

<source id="javax.servlet.http.Cookie.getComment()">
    <category>Cookie Poisoning</category>
</source>

<source id="javax.servlet.http.Cookie.getDomain()">
    <category>Cookie Poisoning</category>
</source>

<source id="javax.servlet.http.Cookie.getName()">
    <category>Cookie Poisoning</category>
</source>

<!-- Information Leakage -->

<source id="java.sql.ResultSet.getString(int)">
    <category>Information Leakage</category>
</source>

<source id="java.sql.ResultSet.getString(String)">
    <category>Information Leakage</category>
</source>

<source id="java.sql.ResultSet.getObject(int)">
    <category>Information Leakage</category>
</source>

<source id="java.sql.ResultSet.getObject(String)">
    <category>Information Leakage</category>
</source>
```

```xml
    <!-- Untrusted Source -->

    <source id="javax.swing.text.TextComponent.getText()">
        <package>javax.swing.text</package>
        <class>JTextComponent</class>
        <method>getText</method>
        <category>Untrusted Source</category>
    </source>

    <source id="javax.swing.text.TextComponent.getSelectedText()">
        <package>javax.swing.text</package>
        <class>JTextComponent</class>
        <method>getSelectedText</method>
        <category>Untrusted Source</category>
    </source>

    <source id="java.io.BufferedReader.readLine()">
        <package>java.io</package>
        <class>BufferedReader</class>
        <method>readLine</method>
        <category>Untrusted Source</category>
    </source>

    <source id="java.io.Reader.read()">
        <package>java.io</package>
        <class>Reader</class>
        <method>read</method>
        <category>Untrusted Source</category>
    </source>

    <source id="javax.servlet.http.Cookie.getValue()">
        <package>javax.servlet.http</package>
        <class>Cookie</class>
        <method>getValue</method>
        <category>Untrusted Source</category>
    </source>

    <source id="javax.servlet.http.HttpServletRequest.getParame-
ter(String)">
        <package>javax.servlet.http</package>
        <class>HttpServletRequest</class>
        <method>getParameter</method>
        <category>Untrusted Source</category>
    </source>

    <source id="java.io.FileInputStream.read()">
        <package>java.io</package>
        <class>FileInputStream</class>
        <method>read</method>
        <category>Untrusted Source</category>
    </source>

    <source id="java.io.InputStreamReader.read()">
        <package>java.io</package>
        <class>InputStreamReader</class>
        <method>read</method>
        <category>Untrusted Source</category>
    </source>
    <source id="java.io.BufferedReader.read()">
        <package>java.io</package>
        <class>BufferedReader</class>
        <method>read</method>
        <category>Untrusted Source</category>
    </source>

    <source id="java.util.Properties.getProperty(String)">
        <package>java.util</package>
        <class>Properties</class>
        <method>getProperty</method>
        <category>Untrusted Source</category>
    </source>

    <source id="java.lang.System.getProperty(String)">
        <package>java.lang</package>
        <class>System</class>
        <method>getProperty</method>
        <category>Untrusted Source</category>
    </source>

    <source id="java.lang.System.getEnv(String)">
        <package>java.lang</package>
        <class>System</class>
        <method>getEnv</method>
        <category>Untrusted Source</category>
    </source>
</sources>
```

Sink Rules

```xml
<?xml version="1.0" encoding="UTF-8"?>

<sinks>
    <!-- SQL Injections -->
    <sink id="java.sql.Statement.executeUpdate(String)">
        <package>java.sql</package>
        <class>Statement</class>
        <method>executeUpdate</method>
        <paramCount>1</paramCount>
        <vulnParam>0</vulnParam>
        <category>SQL Injection</category>
    </sink>

    <sink id="java.sql.Statement.executeUpdate(String,int)">
        <package>java.sql</package>
        <class>Statement</class>
        <method>executeUpdate</method>
        <paramCount>2</paramCount>
        <vulnParam>0</vulnParam>
        <category>SQL Injection</category>
    </sink>

    <sink id="java.sql.Statement.executeUpdate(String sql, int[])">
        <package>java.sql</package>
        <class>Statement</class>
        <method>executeUpdate</method>
        <paramCount>2</paramCount>
        <vulnParam>0</vulnParam>
        <category>SQL Injection</category>
    </sink>

    <sink id="java.sql.Statement.executeUpdate(String sql, String[])">
        <package>java.sql</package>
        <class>Statement</class>
        <method>executeUpdate</method>
        <paramCount>2</paramCount>
        <vulnParam>0</vulnParam>
        <category>SQL Injection</category>
    </sink>

    <sink id="java.sql.Statement.executeQuery(String)">
        <package>java.sql</package>
        <class>Statement</class>
        <method>executeQuery</method>
        <paramCount>1</paramCount>
        <vulnParam>0</vulnParam>
        <category>SQL Injection</category>
    </sink>

    <sink id="java.sql.Statement.execute(String)">
        <package>java.sql</package>
        <class>Statement</class>
        <method>execute</method>
        <paramCount>1</paramCount>
        <vulnParam>0</vulnParam>
        <category>SQL Injection</category>
    </sink>
```

```
<sink id="java.sql.Statement.execute(String,int)">
    <package>java.sql</package>
    <class>Statement</class>
    <method>execute</method>
    <paramCount>2</paramCount>
    <vulnParam>0</vulnParam>
    <category>SQL Injection</category>
</sink>

<sink id="java.sql.Statement.execute(String,String[])">
    <package>java.sql</package>
    <class>Statement</class>
    <method>execute</method>
    <paramCount>2</paramCount>
    <vulnParam>0</vulnParam>
    <category>SQL Injection</category>
</sink>

<sink id="java.sql.Statement.execute(String,int[])">
    <package>java.sql</package>
    <class>Statement</class>
    <method>execute</method>
    <paramCount>2</paramCount>
    <vulnParam>0</vulnParam>
    <category>SQL Injection</category>
</sink>

<sink id="java.sql.Statement.addBatch(String)">
    <package>java.sql</package>
    <class>Statement</class>
    <method>addBatch</method>
    <paramCount>1</paramCount>
    <vulnParam>0</vulnParam>
    <category>SQL Injection</category>
</sink>

<sink id="java.sql.Connection.prepareStatement(String)">
    <package>java.sql</package>
    <class>Connection</class>
    <method>prepareStatement</method>
    <paramCount>1</paramCount>
    <vulnParam>0</vulnParam>
    <category>SQL Injection</category>
</sink>

<sink id="java.sql.Connection.prepareStatement(String,int)">
    <package>java.sql</package>
    <class>Connection</class>
    <method>prepareStatement</method>
    <paramCount>2</paramCount>
    <vulnParam>0</vulnParam>
    <category>SQL Injection</category>
</sink>

<sink id="java.sql.Connection.prepareStatement(String,int[])">
    <package>java.sql</package>
    <class>Connection</class>
    <method>prepareStatement</method>
    <paramCount>2</paramCount>
    <vulnParam>0</vulnParam>
    <category>SQL Injection</category>
</sink>
```

```
<sink id="java.sql.Connection.prepareStatement(String,int,int)">
    <package>java.sql</package>
    <class>Connection</class>
    <method>prepareStatement</method>
    <paramCount>3</paramCount>
    <vulnParam>0</vulnParam>
    <category>SQL Injection</category>
</sink>

<sink id="java.sql.Connection.prepareStatement(String,int,int,int)">
    <package>java.sql</package>
    <class>Connection</class>
    <method>prepareStatement</method>
    <paramCount>4</paramCount>
    <vulnParam>0</vulnParam>
    <category>SQL Injection</category>
</sink>

<sink id="java.sql.Connection.prepareStatement(String,String[])">
    <package>java.sql</package>
    <class>Connection</class>
    <method>prepareStatement</method>
    <paramCount>2</paramCount>
    <vulnParam>0</vulnParam>
    <category>SQL Injection</category>
</sink>

<sink id="java.sql.Connection.prepareCall(String)">
    <package>java.sql</package>
    <class>Connection</class>
    <method>prepareCall</method>
    <paramCount>1</paramCount>
    <vulnParam>0</vulnParam>
    <category>SQL Injection</category>
</sink>

<sink id="java.sql.Connection.prepareCall(String,int,int)">
    <package>java.sql</package>
    <class>Connection</class>
    <method>prepareCall</method>
    <paramCount>3</paramCount>
    <vulnParam>0</vulnParam>
    <category>SQL Injection</category>
</sink>

<sink id="java.sql.Connection.javax.servlet(String,int,int,int)">
    <package>java.sql</package>
    <class>Connection</class>
    <method>prepareCall</method>
    <paramCount>4</paramCount>
    <vulnParam>0</vulnParam>
    <category>SQL Injection</category>
</sink>
```

```
<!-- HTTP Response splitting -->

    <sink id="javax.servlet.http.HttpServletResponse.sendRedi-
rect(String)">
        <package>javax.servlet.http</package>
        <class>HttpServletResponse</class>
        <method>sendRedirect</method>
        <paramCount>1</paramCount>
        <vulnParam>0</vulnParam>
        <category>HTTP Response Splitting</category>
    </sink>

    <sink id="javax.servlet.http.HttpServletResponse.setHeader(String,
String)">
        <package>javax.servlet.http</package>
        <class>HttpServletResponse</class>
        <method>setHeader</method>
        <paramCount>2</paramCount>
        <vulnParam>0</vulnParam>
        <category>HTTP Response Splitting</category>
    </sink>

    <!-- Cross-site Scripting -->
    <sink id="javax.servlet.ServletOutputStream.print(String)">
        <package>javax.servlet</package>
        <class>ServletOutputStream</class>
        <method>print</method>
        <paramCount>1</paramCount>
        <vulnParam>0</vulnParam>
        <category>Cross-site Scripting</category>
    </sink>

    <sink id="javax.servlet.ServletOutputStream.println(String)">
        <package>javax.servlet</package>
        <class>ServletOutputStream</class>
        <method>println</method>
        <paramCount>1</paramCount>
        <vulnParam>0</vulnParam>
        <category>Cross-site Scripting</category>
    </sink>

    <sink id="javax.servlet.jsp.JspWriter.print(String)">
        <package>javax.servlet.jsp</package>
        <class>JspWriter</class>
        <method>print</method>
        <paramCount>1</paramCount>
        <vulnParam>0</vulnParam>
        <category>Cross-site Scripting</category>
    </sink>

    <sink id="javax.servlet.jsp.JspWriter.println(String)">
        <package>javax.servlet.jsp</package>
        <class>JspWriter</class>
        <method>println</method>
        <paramCount>1</paramCount>
        <vulnParam>0</vulnParam>
        <category>Cross-site Scripting</category>
    </sink>
```

```
<sink id="java.io.PrintWriter.print(String)">
        <package>java.io</package>
        <class>PrintWriter</class>
        <method>print</method>
        <paramCount>1</paramCount>
        <vulnParam>0</vulnParam>
        <category>Cross-site Scripting</category>
    </sink>

    <sink id="java.io.PrintWriter.println(String)">
        <package>java.io</package>
        <class>PrintWriter</class>
        <method>println</method>
        <paramCount>1</paramCount>
        <vulnParam>0</vulnParam>
        <category>Cross-site Scripting</category>
    </sink>

    <sink id="javax.servlet.http.HttpServletResponse.sendError(int,
String)">
        <package>javax.servlet.http</package>
        <class>HttpServletResponse</class>
        <method>sendError</method>
        <paramCount>2</paramCount>
        <vulnParam>0</vulnParam>
        <category>Cross-site Scripting</category>
    </sink>

    <sink id="javax.servlet.jsp.JspWriter.write(String)">
        <package>javax.servlet.jsp</package>
        <class>JspWriter</class>
        <method>write</method>
        <paramCount>1</paramCount>
        <vulnParam>0</vulnParam>
        <category>Cross-site Scripting</category>
    </sink>

    <!-- Path Traversal -->

    <sink id="java.io.File(String)">
        <package>java.io</package>
        <class>File</class>
        <method>File</method>
        <paramCount>1</paramCount>
        <vulnParam>0</vulnParam>
        <category>Path Traversal</category>
    </sink>

    <sink id="java.io.RandomAccessFile(String,String)">
        <package>java.io</package>
        <class>RandomAccessFile</class>
        <method>RandomAccessFile</method>
        <paramCount>2</paramCount>
        <vulnParam>0</vulnParam>
        <category>Path Traversal</category>
    </sink>
```

```
 <sink id="java.io.FileReader(String)">
    <package>java.io</package>
    <class>FileReader</class>
    <method>FileReader</method>
    <paramCount>1</paramCount>
    <vulnParam>0</vulnParam>
    <category>Path Traversal</category>
</sink>

<sink id="java.io.FileInputStream(String)">
    <package>java.io</package>
    <class>FileInputStream</class>
    <method>FileInputStream</method>
    <paramCount>1</paramCount>
    <vulnParam>0</vulnParam>
    <category>Path Traversal</category>
</sink>

<sink id="java.io.FileWriter(String)">
    <package>java.io</package>
    <class>FileWriter</class>
    <method>FileWriter</method>
    <paramCount>1</paramCount>
    <vulnParam>0</vulnParam>
    <category>Path Traversal</category>
</sink>

<sink id="java.io.FileOutputStream(String)">
    <package>java.io</package>
    <class>FileOutputStream</class>
    <method>FileOutputStream</method>
    <paramCount>1</paramCount>
    <vulnParam>0</vulnParam>
    <category>Path Traversal</category>
</sink>

<!-- Reflection Injection -->

<sink id="java.lang.Class.forName(String)">
    <package>java.lang</package>
    <class>Class</class>
    <method>forName</method>
    <paramCount>1</paramCount>
    <vulnParam>0</vulnParam>
    <category>Reflection Injection</category>
</sink>

<!-- Command Injection -->

<sink id="java.lang.Runtime.exec(String)">
    <package>java.lang</package>
    <class>Runtime</class>
    <method>exec</method>
    <paramCount>1</paramCount>
    <vulnParam>0</vulnParam>
    <category>Command Injection</category>
</sink>
```

```
<sink id="java.lang.Runtime.exec(String[])">
    <package>java.lang</package>
    <class>Runtime</class>
    <method>exec</method>
    <paramCount>1</paramCount>
    <vulnParam>0</vulnParam>
    <category>Command Injection</category>
</sink>

<sink id="java.lang.Runtime.exec(String,String[])">
    <package>java.lang</package>
    <class>Runtime</class>
    <method>exec</method>
    <paramCount>2</paramCount>
    <vulnParam>0</vulnParam>
    <category>Command Injection</category>
</sink>

<sink id="java.lang.Runtime.exec(String[],String[])">
    <package>java.lang</package>
    <class>Runtime</class>
    <method>exec</method>
    <paramCount>2</paramCount>
    <vulnParam>0</vulnParam>
    <category>Command Injection</category>
</sink>

<sink id="java.lang.Runtime.exec(String[],String[],File)">
    <package>java.lang</package>
    <class>Runtime</class>
    <method>exec</method>
    <paramCount>3</paramCount>
    <vulnParam>0</vulnParam>
    <category>Command Injection</category>
</sink>

<sink id="java.lang.System.load(String)">
    <package>java.lang</package>
    <class>System</class>
    <method>load</method>
    <paramCount>1</paramCount>
    <vulnParam>0</vulnParam>
    <category>Command Injection</category>
</sink>


<sink id="java.lang.System.loadLibrary(String)">
    <package>java.lang</package>
    <class>System</class>
    <method>loadLibrary</method>
    <paramCount>1</paramCount>
    <vulnParam>0</vulnParam>
    <category>Command Injection</category>
</sink>
```

```
    <!-- XPath Injection -->

    <sink id="javax.xml.xpath.XPath.compile(String)">
        <package>javax.xml.xpath</package>
        <class>XPath</class>
        <method>compile</method>
        <paramCount>1</paramCount>
        <vulnParam>0</vulnParam>
        <category>XPath Injection</category>
    </sink>

    <sink id="javax.xml.xpath.XPath.evaluate(String,InputSource)">
        <package>javax.xml.xpath</package>
        <class>XPath</class>
        <method>evaluate</method>
        <paramCount>2</paramCount>
        <vulnParam>0</vulnParam>
        <category>XPath Injection</category>
    </sink>

    <sink id="javax.xml.xpath.XPath.evaluate(String,InputSource,QName)">
        <package>javax.xml.xpath</package>
        <class>XPath</class>
        <method>evaluate</method>
        <paramCount>3</paramCount>
        <vulnParam>0</vulnParam>
        <category>XPath Injection</category>
    </sink>
    <sink id="javax.xml.xpath.XPath. evaluate(String,Object)">
        <package>javax.xml.xpath</package>
        <class>XPath</class>
        <method>evaluate</method>
        <paramCount>2</paramCount>
        <vulnParam>0</vulnParam>
        <category>XPath Injection</category>
    </sink>

    <sink id="javax.xml.xpath.XPath.evaluate(String,Object,QName)">
        <package>javax.xml.xpath</package>
        <class>XPath</class>
        <method>evaluate</method>
        <paramCount>3</paramCount>
        <vulnParam>0</vulnParam>
        <category>XPath Injection</category>
    </sink>

    <sink id="org.apache.xpath.XPath(String,SourceLocator,PrefixRe-
solver,int)">
        <package>org.apache.xpath</package>
        <class>XPath</class>
        <method>XPath</method>
        <paramCount>4</paramCount>
        <vulnParam>0</vulnParam>
        <category>XPath Injection</category>
    </sink>
    <!-- LDAP Injection -->

    <sink id="com.novell.ldap.LDAPConnection.connect(String,int)">
        <package>com.novell.ldap</package>
        <class>LDAPConnection</class>
        <method>connect</method>
        <paramCount>2</paramCount>
        <vulnParam>0</vulnParam>
        <category>LDAP Injection</category>
    </sink>

    <sink id="com.novell.ldap.LDAPConnec-
tion.search(String,int,String,String[],boolean)">
        <package>com.novell.ldap</package>
        <class>LDAPConnection</class>
        <method>search</method>
        <paramCount>5</paramCount>
        <vulnParam>0</vulnParam>
        <category>LDAP Injection</category>
    </sink>

    <sink id="com.novell.ldap.LDAPConnec-
tion.search(String,int,String,String[],boolean,LDAPSearchQueue)">
        <package>com.novell.ldap</package>
        <class>LDAPConnection</class>
        <method>search</method>
        <paramCount>6</paramCount>
        <vulnParam>0</vulnParam>
        <category>LDAP Injection</category>
    </sink>

    <sink id="com.novell.ldap.LDAPConnec-
tion.search(String,int,String,String[],boolean,LDAPSearchQueue)">
        <package>com.novell.ldap</package>
        <class>LDAPConnection</class>
        <method>search</method>
        <paramCount>6</paramCount>
        <vulnParam>2</vulnParam>
        <category>LDAP Injection</category>
    </sink>

    <sink id="javax.naming.directory.DirContext.search(String name, At-
tributes matchingAttributes)">
        <package>javax.naming.directory</package>
        <class>DirContext</class>
        <method>search</method>
        <paramCount>2</paramCount>
        <vulnParam>0</vulnParam>
        <category>LDAP Injection</category>
    </sink>

    <sink id="javax.naming.directory.DirContext.getSchema(String name)">
        <package>javax.naming.directory</package>
        <class>DirContext</class>
        <method>getSchema</method>
        <paramCount>1</paramCount>
        <vulnParam>0</vulnParam>
        <category>LDAP Injection</category>
    </sink>
```

بسم الله الرحمن الرحيم
١٤١٨

إختبار أمن تطبيقات الويب باستخدام الوكيل البرمجي

محمد عمران

بحث مقدم لنيل درجة الماجستير في العلوم (علوم الحاسبات)

إشراف

أ. د. فتحي البرعي عيسى

قسم علوم الحاسب
كلية الحاسبات وتقنية المعلومات
جامعة الملك عبدالعزيز
جدة – المملكة العربية السعودية

جمادى الآخرة 1435هـ - ابريل 2014م

إختبار أمن تطبيقات الويب باستخدام الوكيل البرمجي

محمد عمران

المستخلص

مع التقدم في التكنولوجيا يتم نشر المزيد والمزيد من الخدمات والعمليات في الشبكة العالمية. هذا أدى الى زيادة الطلب على تطوير واستخدام تطبيقات الويب وبالتالي زيادة عدد الهجمات على شبكة الإنترنت و نقاط الضعف الأمنية في تطبيقات الويب . حدث بالمقابل تطورا في مجال أمن تطبيقات الويب في السنوات السابقة ، إلا أن نقاط الضعف ما زالت موجوده بشكل او باخر وأساليب الهجوم لم تتوقف يوما من التطور. هذا من شانه أوجد فرصة اقوى للتقنيات و الادوات المستخدمه لكشف ومنع الهجمات الأمنية من الحدوث.

من ضمن العديد من الفئات والأصناف، يوجد صنف من ثغرات الضعف الامنية الموجودة في الشبكات يعرف باسم التاكد من مدخلات المستخدم والتي تستخدم بشكل غير صحيح حيث يبدا الهجوم بالمدخل الغير سليم وينتهي بتنفيذ ثغرة امنية حساسة.

في هذه الأطروحة، تم تقديم تقنية تحليل ساكنه ودينامكية متكاملة (مستنده على نظام العميل) للكشف عن نقاط الضعف الأمنية في التطبيقات المستندة إلى الويب المكتوبة بلغة الجافا. في البداية تم تحليل العيب الساكن بتتبع انتشار إدخال المستخدم في البرنامج والذي ساعد في الكشف عن نقاط الضعف في التعليمات البرمجية للمكتوبة بلغة المصدر. في المرحلة التالية ، تم استخدام التحليل الديناميكي لمنع الهجمات والحد من الايجابيات الكاذبة الناتجة عن التتحليل الساكن. التقنية التي استخدمت في دراستنا هي امتدا للكشف عن الثغرات الموجودة في نفس الصنف ومكتوبة بلغة برمجيه أخرى موجهة نحو الهدف. أخيرا، تم بناءالتقنية المقترحه كأداة اختبار أمنية (مستنده على نظام العميل) تدعى مختبر نقاط الضعف الامنية المبني على الجافا (JVT).