國立臺灣大學管理學院資訊管理系

碩士論文

Department of Information Management

College of Management

National Taiwan University

Master Thesis

網站應用程式安全性弱點分析方法與工具之研究

A Study of Methods and Tools for Analyzing Security Vulnerabilities in Web Applications

蔡依珊

Yi-Shan Tsai

指導教授：蔡益坤 博士

Advisor: Yih-Kuen Tsay, Ph.D.

中華民國 98 年 7 月

July, 2009

網站應用程式安全性弱點分析方法與工具之研究

A Study of Methods and Tools for Analyzing Security Vulnerabilities in Web Applications

本論文係提交國立臺灣大學
資訊管理研究所作為完成碩士學位
所需條件之一部分

研究生：蔡依珊 撰

中華民國九十八年七月

謝辭

妳好，很高興妳對我的致謝辭感到興趣。

沒想到自己也有寫致謝辭的一天，因為通常這種人都是發表書籍或投稿論文的人才會做的事，所以我想，我也算是做了一件偉大的事了吧。

還記得剛進臺大資管所的自我介紹中"對未來的期望"寫著:畢業後能帶走專業與許多回憶。現在的我就是當時認為的未來的我，"未來的期待"也變成了"既有的事實"。從不斷付出但不求回報的家人、老師和同學身上得到了許多寶藏，不只是專業知識，還有做人處事的道理，雖然這樣寫好像很八股，但，這就是讓我的生命、我的生活富有的原因。

一路上，遇到的貴人很多，要感謝的人也很多。請容許我用列舉的方式寫出:

爸爸、媽媽、爺爺、奶奶、外公、外婆、大伯、大伯母、姑媽、姑丈、大阿姨、大姨丈、小阿姨、小姨丈、依珍、大哥、大嫂、二哥、二嫂、大姐、大姐夫、二姐、小表姐;宋雲喬、叔叔、阿姨、姑姑、姑丈;阿詮的爸爸、阿詮的媽媽;古芳宜老師、林雅盛老師、李長伯老師、毛成惠老師、溫嫩玫老師、Curtis、劉興、古政元老師、吳帆老師、吳榮訓老師、蔡益坤老師、莊浴澤老師、Paul;佳慧、螃蟹、Paula、Silas、林怡君、文琇、小球、小宛、妍萱、趙香腸、咻、小臉、廖慧儒、阿Q、吳家宜、陳君瑋、monokol、陳建州、瀉遜、盈慧、阿良、聖祐、張晉碩、常怡文、屠敬傑、鍾正一、蔡明憲、郁方、楊德威、小p、葉睿元、游昇峰、戴智斌。

尤其感謝長久陪伴也支持我每個決定的爸爸和媽媽。我愛你們非常非常。

願我愛的人們能夠健健康康、順順利利;願未來的我能夠再度實現不可能的夢想。

蔡依珊　謹識

于臺灣大學資訊管理研究所

民國九十八年七月

論文摘要

學生: 蔡依珊　　民國九十八年七月
指導教授: 蔡益坤

網站應用程式安全性弱點分析方法與工具之研究

身為全球經濟體系基礎建設中的一部分，網站應用程式提供了一個虛擬平台做為使用者之間的溝通橋樑，這使得其地位顯得相當重要。然而，網路安全漏洞的問題卻日益嚴重，並對網站應用程式的發展造成了負面的影響。在應用程式的開發過程中，網站應用程式源碼檢測可做為解決此項問題的其中一項方法。但是人工檢測程式源碼過程費時、費力或因人為疏失而導致不精確的檢測結果；再加上檢測程式源碼人員必須具備資訊安全的專業知識背景。因此，自動化源碼檢測工具的需求，也就因應而生。早期自動化方法與工具僅應用在軟體應用程式上，而後才延伸至網站應用程式，但目前來說，評估靜態工具與方法精確性之研究也較少。換句話說，靜態工具開發者在沒有與其他工具比較之情況下宣稱其靜態方法與工具具有效率與有效性就失去了說服力。

本篇論文目的在於評估現有四個靜態分析方法與工具之精確性，為此我們設計了一套含有安全漏洞的程式源碼之標準檢查程式(例如跨站腳本攻擊與資料庫安全漏洞的注入)，且標準檢查程式內也含有不同的資料結構與控制流程敘述。更明確地說，透過我們設計的標準檢查程式來評估現有靜態方法與工具之效能，並以統計數據方式呈現工具間於特定安全漏洞類別之精確的處理程度。最後，我們整合這四個靜態分析方法與工具之結果，找出現有靜態方法與工具不足之處，以協助未來靜態方法與工具之開發。

關鍵字: 安全漏洞，網站應用程式，精確性，標準檢查程式，程式源碼檢測，靜態分析方法與工具。

THESIS ABSTRACT

Graduate Institute of Information Management
National Taiwan University

Student: Tsai, Yi-Shan
Advisor: Tsay, Yih-Kuen

Month/Year: July, 2009

A Study of Methods and Tools for Analyzing Security Vulnerabilities in Web Applications

As part of the infrastructure of the global economy, Web applications are of the utmost importance because they provide a virtual space where end users can communicate with one another. A negative aspect of this development is that the number of security vulnerabilities is growing constantly. One method used to solve such problems involves reviewing program code as a part of the development process. However, manual code verification is time-consuming, error-prone, and costly; and code auditors need a security background in order to audit the code. Thus, there is an urgent need for automated solutions to check whether Web applications are vulnerable. Verification tools have long implemented analysis methods in software applications and Web applications, but little research has been performed to evaluate the efficacy of each tool. Of course, developers claim that their tools are effective and efficient, but they do not compare their tool with others.

In this thesis, our objective is to evaluate the efficacy of existing verification tools. To this end, we build benchmark cases of vulnerable code that may cause security problems, such as cross-site scripting and SQL injection, but some benchmark cases do not consist of vulnerable code to determine if a false positive occurs after the tool scans the code. Specifically, we use the developed benchmark cases to test four static analysis tools that generate reports of vulnerable program locations, and evaluate the performance of the tools statistically. Moreover, the benchmark cases enable us to identify the structures or control flow statements that cause false alarms in the four tools. As a result, we can determine which benchmark cases are not handled in the target tools.

Keywords: Security Vulnerabilities, Web Applications, Precision, Benchmark, False Alarm, Code-Verification, Static Analysis Tools.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

As part of the infrastructure of the global economy, online systems are of the utmost importance because they provide a virtual space where end users can communicate and deal with business matters. Static Web pages can not provide dynamic content to satisfy the demands of clients, employees, and business partners around the world. Consequently, dynamic Web pages have become the de facto standard for providing online services ranging from message boards to online trading systems, such as shopping and banking applications. Moreover, ubiquitous Web applications, implemented with server-side scripting language or client-side scripting language, handle sensitive data and perform critical tasks. For example, a user can log into an online banking system anytime anywhere and interact with a Web database application that generates outputs dynamically.

Unfortunately, the convenience and ubiquity of online applications have given rise to an increasing number of security problems. For example, in 2005, the student log-in Web page of a major university was found to be vulnerable and was leaking applicants' personal information [35]. In another example, in 2007, a recruiting Web page divulged the personal information of 1.3 million job seekers [41]. Obviously, the overheads of organizations whose Web sites have been attacked will increase because potential victims must be notified that their identities may have been stolen. Moreover, the reputation of such organizations will be damaged, and the organizations may even be accused of carelessness. According to a recent report, 80 percent of phishing attacks, which try to collect confidential information by copying or spoofing genuine Web sites for pecuniary

benefits, occur in the financial services industry. The problem is becoming increasingly serious, as shown by the following statistics: in the first half of 2007, there were 32,939 attacks compared to 97,763 in the second half of the year [40].

Many Web applications are infected with malicious strings in the parameters used for sensitive operations because external files or user inputs have not been properly checked or sanitized. The strings generate unintended outputs or destroy the structure of Web applications, and thereby benefit attackers. Dynamic content generated by external user inputs or databases may be susceptible to security vulnerabilities, such as cross-site scripting and SQL injection. Survey-based statistics reported in [40] show that cross-site scripting attacks nearly doubled in 2007 (from 6,961 to 11,253). One major problem with detecting vulnerabilities is that Web application developers are under time-to-market pressures, or they do not know the potential for incorrect usage of each scripting language and the available sanitization functions.

1.2 Motivation and Objectives

Verification tools have long implemented analysis methods in software applications [28] and Web applications. Because of the security problems that arise with Web applications, static analysis methods and tools are very important. One of the major purposes of static analysis methods and tools is to alleviate the load of dynamic analysis at run time. Basically, vulnerability checkers use data flow analysis to track information flows from external resources of Web applications, such as user inputs (also called sources) to program lines where sensitive functions (also called sinks) are located. There are two types of static analysis methods: program-code checking and user-input detecting. Program-code checking is executed before run time of Web applications to identify the line of program code in which vulnerabilities occur. It is estimated that the cost of using static analysis tools is 17% less than using human checkers [1]. User-input detecting, on the other hand, is implemented at run time, and an analyzer verifies if the user-supplied data transferred to sensitive functions is vulnerable. Thus, static analysis for security vulnerabilities should be applied routinely in the development process.

However, very little research has been carried out to establish benchmark cases con-

taining vulnerable sinks and to evaluate the precision of each tool. Of course, developers claim that their tools are effective and efficient, but they do not compare them with other tools. In this thesis, we test four verification tools with the developed benchmark cases to determine which benchmark cases are not handled in the target tools, and improve research on static analysis methods for security. The effectiveness of sanitization functions is another research area that has not been widely investigated. We therefore evaluate if sanitization functions are effective by inserting tainted data into Web applications at run time to check the effectiveness of sanitization operations.

1.3 Thesis Outline

The rest of the thesis is structured as follows:

- In Chapter 2, we provide preliminaries of this thesis: Web architecture, common vulnerabilities in Web applications, regular expressions.
- In Chapter 3, we design a test suite to implement available static analysis tools. We display representative benchmark cases according to vulnerability categories in this chapter, and all benchmark cases with program code and description are attached in the Appendix section.
- In Chapter 4, we implement four static analysis tools with a benchmark into five coarser categories of security vulnerabilities, and analyze the result generated by these tools via statistics formulae. Therefore, we know which part is not enough in particular vulnerability categories.
- In Chapter 5, we present earlier research in static analysis methods for security vulnerabilities in Web applications and a supplement method verifying the effectiveness of sanitization functions reported by static analysis.
- In Chapter 6, we write a conclusion to illustrate contributions and future work.

Chapter 2

Preliminaries

2.1 Web Architecture

The client-server computing model, where applications do not run completely on a central computer or on a workstation, discriminates client applications from server applications. A client initiates requests, waits for or receives replies, connects to several servers, and interacts with other end users with a graphic user interface (GUI), while a server waits for or replies to requests sent from connected clients and transfers data to intended clients. Specifically, a server, mostly, is separated into two portions: an application server running business logics and a database server to save data supporting the operation of a server.

2.1.1 Web Applications

In software engineering, a Web application is an application accessed with Web browsers, for instance, Internet Explorer and Mozilla Firefox enabling users to interact with texts, images, videos and animations located on the World Wide Web, coded by such server-side scripting languages as PHP, JSP or ASP.

Web applications are increasingly popular due to the feature of ubiquity, the abilities for updating and maintaining Web applications , mostly, without installing more complex software on a client's local computer, sometimes called a thin client, a client computer relies mainly on the remote server to process activities in the client-server architecture model. Common Web applications rather than desktop applications display in a whole lot of variations, for example, Google Calendar, Webmail, wikis, and online auctions. However, the convenience of Web applications comes with the price that leads to security

issues through transforming data, for instance, cookie files stored at client side and session files saved at server side.

2.1.2 Security Problems in PHP

Normally when a Web browser requests a static or dynamic Web page with HTTP requests, the Web server returns HTTP responses rendered into Web-style pages. However, if the Web page contains several server-side scripts, the Web server executes the scripts before an HTTP response is sent back to the Web browser. Function of server-side scripts are expected to edit dynamic content of a Web page which is customized for users' favorites or habits.

Although Web applications can be implemented by many programming languages [12, 23, 20], we use PHP in the following examples due to three reasons: firstly, nowadays many researchers describe security vulnerabilities in Web applications with PHP. Secondly, it is suitable to take one server-side scripting language for maintaining the consistence and integrity. Thirdly, the proportion of Web applications coded in PHP rise increasingly, more than a third of security vulnerabilities, amounted to [6]: 12 % in 2003, 20 % in 2004, 28 % in 2005, 43 % in 2006, 36 % in 2007, 34.8 % in 2008, and 37.1 % in the first quarter of 2009.

For the popularity of PHP, Netcraft April 2007 report [10] mentions that over 20 million domains coded in PHP are on the Web, TIOBE [37] publishes PHP is the fourth in developer popularity, and SYHUNT [39] research states vulnerabilities in PHP occupy 89 percentage. As the usage of PHP is on the rise, Web sites in PHP become favorite targets of attackers. Natural PHP gives rise to a number of vulnerabilities and exploits spreading wildly by attackers, but these Web sites use sensitive functions, such as fopen(), exec(), and require() to develop a dynamic application that may be executed malicious code. PHP seems a common tool for evil, but not only developers but clients, Webhosts, administrators, and security analyzers ought to get together to solve security problems.

2.2 Common Vulnerabilities in Web Applications

In this section, we introduce seven common vulnerabilities in Web applications, namely: cross-site scripting, injection flaws, malicious file execution, cross site request forgery, HTTP response splitting, resource injection, and information leakage selected from Open Web Application Security Project (OWASP), a community includes corporations, educational organizations, and individuals from all over the world and works to create freely security-based documentations, methodologies, tools, and technologies. These seven security vulnerabilities are related to server side where we study analysis methods and tools to verify server-side scripting languages.

2.2.1 Cross-Site Scripting (XSS)

Cross-site scripting (CWE-79 [7]) occurs when a Web application takes inputs from users via Web browsers without validating or encoding content, allowing attackers executing scripts at victims' Web browsers, thus hijacking user cookies, defacing Web sites, or introducing worms. There are three kinds of cross-site scripting as follows, and attackers are expected to combine these three types:

- **DOM-based cross-site scripting:** (also called Type 0 cross-site scripting) The JavaScript code and variables of a site are manipulated through HTML elements including:

  document.URL: The URL property returns the URL of a current document.

  document.location: The location property returns the URL of a current document.

  document.referrer: The referrer property returns the URL of a document referred to by a current document.

  window.location: The location property returns a location object containing information about the URL of a document ,and provides methods for changing URL.

- **Reflected cross-site scripting:** (also called type 1 or non-persistent cross-site scripting) It is the most common type, and occurs when data provided from user

inputs is used promptly by server-side scripts to introduce a dynamic Web page back to that Web user. An example of reflected cross-site scripting is as follows:

```
echo $_REQUEST['userinput'];
```

- **Stored cross-site scripting:** (also called type 2 or persistent cross-site scripting) Hostile data provided by attackers is stored in such back-end systems as a file and a database, and is displayed, usually a tainted hyperlink, in trusted Web sites, for instance, blogs, forums, emails, and content management system (CMS), a computer application used to add content in order to keep the newest information among group members, where victims click hostile hyperlinks and fall into attackers' traps.

2.2.2 SQL Injection

Injection Flaws, especially SQL injection (CWE-89 [7]), occur when inputs are sent to an interpreter as a fragment of command or database query. User inputs are incorrectly filtered out escape characters embedded in SQL statements or are not strongly typed, so that attackers read, update, or delete arbitrary data in Web applications. Two SQL injection examples as follows are brought about by not filtering out escape characters and type-checking from inputs of clients:

- **Inaccurately filter out escape characters:** This type of SQL injection occurs whenever user inputs pass into interpreter without filtering out unfriendly escape characters. An example is as follows:

```
$ID = $_GET['id'];
$query = "SELECT userPSW FROM User WHERE userID = '" + $ID + "';"
mysql_query($query);
```

The SQL statement is made to sieve out a record from the table "User" with specific user identification provided by users. However, the SQL injection occurs when an attacker inserts a hostile value, in the following statement, into the variable **userID**.

```
x' OR '1'='1
```

The SQL statement is rendered by the interpreter, and SQL injection occurs:

SELECT userPSW FROM User WHERE userID = '**x' OR '1'='1**';

- **Inaccurately type manipulating:** This type of SQL injection occurs whenever programmers take user inputs as a part of SQL statements without validating the type of values. We give an example in the following query statement:

```
$count = $_GET['Count'];
$query = "SELECT * FROM online_counter WHERE id = " + $count + ";"
mysql_query($query);
```

We assume the type of variable **count** is numeric and an attacker inserts non-numeric value into **count**, but the interpreter accepts the following value without checking its type:

123;DROP TABLE User

The SQL statement is rendered by the interpreter, and SQL injection occurs:

SELECT * FROM online_counter WHERE id=123;DROP TABLE User;

Unfortunately, sensitive information in database is excavated if an attacker inputs hostile data as a part of SQL statements, but there is a potential risk in Web sites in serious destruction.

2.2.3 Malicious File Execution

Remote file inclusion (RFI) allows attackers to including hostile files instead of data strings in the previous two security vulnerabilities: cross-site scripting and injection flaws. In many Web applications, structures allow the usage of external object references, particularly in PHP, but a common problem, being vulnerable to tamper with values of filename through arguments in URL or to accept filenames inserted by users to include local files which are not published publicly, leads to malicious file execution. An example of malicious file execution is as follows:

```
include $_POST['filename'];
```

A client (as a hostile one) sees "http://www.example.com/test.php" in URL, but then guesses and changes a file name to access, for example, "http://www.example,com/unexpectedpage.php. The difficulty to detect this security vulnerability is that the file name is provided from users at run time, so that program code does not be verified at the stage of static analysis. An intuitive way to solve this problem is to limit the external file names will a white list containing pages you expect.

2.2.4 Cross Site Request Forgery (CSRF)

Cross site request forgery (CWE-352 [7]) comes out whenever an attacker coerces a victim, logging in a normal Web application, into doing some sensitive activities unconsciously. Although CSRF does not directly occur in not validating the server-side scripting language, it indirectly takes place when attackers publish hostile hyperlinks in trusted Web sites or in content of emails, so that lead to execute sensitive functions without users' consciousness. For example, an attacker transfers funds without perception of a victim, logging in an online banking system. The following example is an easy one to present CSRF, but the risk of CSRF in real world is more frightful: the client logs in a Web mail and then unwittingly clicks a malicious hyperlink which contains such content as:

click me

Then a client logging in this Web mail is forced to log out unconsciously.

2.2.5 HTTP Response Splitting

Since Web servers receive requests without verifying values inserted by users, HTTP response splitting (CWE-352 [7]) may occur. HTTP response splitting occurs if Web sites allow attackers writing data into an HTTP header without verification the usage of CR (carriage return, also given by %0d or \r) and LF (line feed, given by %0a or \n) so that attackers can create an entirely new HTTP response. The following code segment reads a parameter **author** from an HTTP request and sets the value of the parameter in a cookie as a part of an HTTP response.

```
01:  ...
02:  $author = $_GET[author];
03:  setcookie("Author", $author);
04:  ...
```

If a parameter contains only allowable alphanumberic characters, such as Tabitha, the HTTP response takes the following form:

```
HTTP/1.1 200 OK
Set-Cookie: Author = Tabitha
...
```

However, HTTP response splitting occurs when user inputs used in HTTP headers are not verified. If parameters consist of malicious values, for example, Hackers Never Die \r \n \r \n HTTP/1.1 200 OK \r \n..., the HTTP response will divided into two responses as follows:

```
HTTP/1.1 200 OK
Set-Cookie: Author = HACKERS Never Die

HTTP/1.1 200 OK
...
```

Users receive an HTTP response according to an HTTP request, but the second response, stored in the cache, will be accepted by the user who sends the next HTTP request. Attackers treat users who receive malicious responses rendered by users' browsers, and cross-site scripting may occur.

However, HTTP response splitting does not occur in PHP, because PHP has fixed the header injection by limiting each header to a single line in PHP version 5.1.2 released from the PHP official Web site since January 12, 2006. PHP has avoided the usage of carrier return and line feed.

2.2.6 Resource Injection

Path traversal (CWE-22 [7]) and external control of assumed-immutable Web parameter (CWE-472 [7]) belong to resource injection (as known as insecure direct object reference in OWASP [30] classification.) Resource injection occurs if a Web server allows users implementing internal objects, such as files or directories without checking parameters

invoked into sensitive functions. For instance, the following program segment allows users removing directory by inserting a directory name they want. Attackers are expected to delete an unexpected directory if they know the directory names or paths.

```
01:  ...
02:  $dirName = $_GET[dirName];
03:  rmdir($dirName);
04:  ...
```

2.2.7 Information Leakage

Information Leakage (as known as information leakage and improper error handling in OWASP [30]) contains information leak (CWE-200 [7]), information leak through debug information (CWE-215 [7]), and error message information leak (CWE-209 [7].) Generally, information leak is divided into two kinds: one is system error message, and the other is user-defined error message. No matter which kind of error message is, developers of Web applications should turn off these pages displaying error messages needed in the developing phase to avoid information divulgation. The following program segment contains function **debug_print_backtrace()** printing a back trace, and function names, the program line, and file names will leak through this function.

```
01:  ...
02:  function f() {
03:    echo "The back trace is ". debug_print_backtrace();
04:  }
05:  ...
```

2.3 Regular Expression

In computing, regular expressions ,abbreviated as regex or regexp, containing two types: the POSIX and the Perl-Derivative regular expression, present flexible and precise meaning for identifying strings, such as particular characters or special patterns, and a regular expression processor, a program either to serve as a parser generator or to examine strings and to identify portions matching provided patterns, interprets regular expression written in a formal language. We can group the characters into the specification as follows:

- Normal characters matching themselves like "hello"
- Start and end indicators: ∧ and $
- Count indicators: =, * and ?
- A logical operator: |
- Grouping symbols: {}, () and []

With PHP scripting language, one of the most common functions of regular expressions is data validation, and PHP contains several validation functions allowing Web applications matching a particular pattern using regular expressions, making sure there is no space or asterisk. An example to validate the form of emails shows as follows:

$$\wedge[\text{a-zA-Z0-9.\_-}]+@[\text{a-zA-Z0-9-}]+[\text{a-zA-Z.}]\{2,5\}\$$

PHP deals with two types of regular expressions, showing the form of syntax at Table 2.1: Perl 5 compatible patterns and POSIX standard compatible patterns. PHP includes seven functions with Perl 5 compatible patterns:

- ereg_replace: Replace regular expression
- ereg: Match regular expression
- eregi_replace: Replace regular expression case-insensitively
- eregi: Match regular expression case-insensitively
- split: Split string into array by regular expression

- spliti: Split string into array by regular expression case-insensitively
- sql_regcase: Make regular expression for case insensitive match

In the other side, PHP contains six functions with POSIX standard compatible patterns:

- preg_grep: Return array entries matching the pattern
- preg_match_all: Perform a global regular expression match
- preg_match: Perform a regular expression match
- preg_quote: Quote regular expression characters
- preg_replace_callback: Take a regular expression search and replace using a ballback
- preg_split: Split strings by a regular expression

Regular Expression (pattern)	**Example (subject)**	**Explanation**
apple	apple pie	the pattern is present anywhere in the subject
∧apple	apple juice	the pattern is present at the beginning of the subject
apple$	red apple	the patter is present at the end of the subject
apple/i	AppLe	the string is in case insensitive mode
∧apple$	apple	the string contains only the "apple"
apple*	appl, applee	zero or more "e" after "appl"
apple+	apple, applee	one or more "e" after "appl"
apple?	apple, apply	zero or one "e" after "appl"
apple{1}	apple	exact one "e" after "appl"
apple{1,}	apple, appleee	one or more "e" after "appl"
apple{2,3}	applee, appleee	either two or three "e" after "appl"
ap(ple)*	ap, appleple	zero or more "ple" after "ap"
apple|pie	apple, pie	either "apple" or "pie"
a.ple	ample, apple	any character in place of the dot
∧.{5}$	apple, kitty	a string with exact five characters
[xyz]	xyz, yyzzx	an "x", "y", or "z" in the string
[a-z]	apple	any lowercase letter in the string
[a-zA-Z]	apple, APPLE, APP13	any lowercase or uppercase letter in the string
[∧aA]	pie	the actual character cannot be an "a" or "A"

Table 2.1: Regular expressions in PHP.

Chapter 3

Benchmark Cases for Evaluating Tools

3.1 Benchmark Overview

Some research [23, 43, 21, 18, 22, 42, 25, 45] has performed static analysis methods with a benchmark (a.k.a. a test suite) downloaded from Web sites providing open source applications, and most of analyzed languages are JAVA and PHP. The difference in this thesis is we build a benchmark instead of downloading from open source Web sites.

The reasons why we design a benchmark are:

- Firstly, we know the structures (the flow of a program) and semantics of benchmark cases so that we decide whether the result produced by static analysis methods or tools is correct.

- Secondly, we add some stranger or more complex data structures in test cases, such as variable variable and string index, but test static analysis methods or tools with benchmark cases of more complicated data structure.

  Variable variable: A variable variable treats the value of a variable as a variable name. Figure 3.1, for example, the value of $$vv at line 4 is propagated from the variable **$taint** at line 2 through the assignment statement at line 3. In other words, the value of **$taint** and **$$vv** is the same.

  String index: An index is made by a string instead of a number. Figure 3.2, for instance, the indices assigned with strings named one and two at line 3.

Except for data structure, we try different functions in benchmark cases as follows:

```
01: <?php
02:   $taint = $_GET['source'];
03:   $vv = 'taint';
04:   echo "taint data is ".$$vv;
05: ?>
```

Figure 3.1: The structure of variable variable.

```
01: <?php
02:   $taint=$_GET["name"];
03:   $myArray=array ("one"=>$taint,"two"=>"fixed data");
04:   $arrindex=array ("one","two");
05:   foreach($arrindex as $val)
06:     echo $myArray[$val]."<br>";
07: ?>
```

Figure 3.2: The structure of string index.

- Display handling: echo() and print_r;
- Control flow statments: while, do-while, and if;
- SQL statement in database manipulation: select, insert, update, and drop;
- File operation: unlink(), chmod(), fopen(), fread(), and rmdir();
- Command execution: exec() and passthru();
- File inclusion: require(), require_once(), include(), and include_once();
- Information leakage: phpinfo(), getMessage(), and debug_print_backtrace();
- Log forging: error_log() and fputs();
- Reflection injection: call_user_func();
- Variable overwrite: parse_str() and mb_parse_str().

3.2 Benchmark Description

We design 90 benchmark cases to implement static analysis tools, but we only show macro benchmark cases in this section. The entirely benchmark cases are displayed in the appendix section, and we describe their operation and which vulnerabilities may occur at the bottom of each of them.

3.2.1 Display Handling

Programs display strings and values of variables through function echo or print_r(). Figure 3.3 will print strings via theses two functions. The result will be printed, at line 3, by function **echo**, and line 4, by function **print_r()**, through the parameter, named **$name** at line 2, got from request method **GET**, and XSS occurs at line 3 or 4 if Web site gets a malicious input.

```
01: <?php
02:   $name = $_GET['name'];
03:   echo "Welcome $name";
04:   print_r ($a);
05: ?>
```

Figure 3.3: Display handling.

3.2.2 Control flow statements

Programs contain control flow statements, such as if statement, while loop, do-while loop, and so on to test whether static analysis tools find vulnerabilities in deeper program locations. Figure 3.4 tests if tools claim cross-site scripting occurs when sensitive sinks wrapped in control flow statements.

3.2.3 SQL Statements in Database Manipulation

SQL injection occurs if a part of an SQL statement provided from users is not verified, and Web sites display unexpected content or are destructed. The function **mysql_query** executes an SQL statement, and its performance is diversified so that we test if tools

```
01: <?php
02:   $db = $_GET['dbName'];
03:
04:   $con = mysql_connect("localhost", "root", "svvrl");
05:   mysql_select_db($db) or die("無此資料庫: $db");
06:   $result = mysql_query("select * from stuInfo where stu_id=100001");
07:
08:   if( $row = mysql_fetch_array($result) ) {
09:     echo $row['stu_name']."<BR>";
10:   }
11:
12:   while ( $row = mysql_fetch_array($result) ) {
13:     echo $row['stu_name']."<BR>";
14:   }
15:
16:   do{
17:     echo $row['stu_name']."<BR>";
18:   }while ( $row = mysql_fetch_array($result));
19: ?>
```

Figure 3.4: Control flow statements.

claim an SQL injection occurs with different of SQL statements. It is an example in Figure 3.5:

3.2.4 File Operation

Many functions are able to manipulate such objects as files or directories. In our test cases, we read and write content of files, upload files, change an authentication level, or remove directories. For example, file manipulation in Figure 3.6 should be claimed a resource injection occurs at line 3 , 4, and 7.

A Web server sometimes writes log files to record situations of users or execution environments. If a program does not contain verification functions to check whether the values furnished by users is vulnerable, the structure of log files may be demolished. A log-recording example is in Figure 3.7, and tools should claim the value carried by variable **$logMsg** at line 11 may cause log-file destruction:

```
01: <?php
02:   $link = mysql_pconnect("localhost", "root", "svvrl");
03:   mysql_select_db("tabitha") or die("無法選擇資料庫");
04:
05:   $query="SELECT stu_id, stu_name, stu_score FROM stuinfo WHERE stu_id='100001'";
06:   $query_result=mysql_query($query);
07:   $result=mysql_fetch_row($query_result);
08:   $source1=$result[2];
09:
10:     if(!empty($source1)){
11:
12:     $query="SELECT stu_score FROM stuinfo WHERE stu_name="".$source1.""";
13:     $query_result = mysql_query($query);
14:   }
15: ?>
```

Figure 3.5: SQL statements in database manipulation.

3.2.5 Command Execution

Developers should avoid designing a command execution for Web users, or Web users are able to execute unexpected sensitive commands. However, if a program contains a command operation, tools should make a warning. In Figure 3.8, we execute command statements at line 3 and 4 through function **exec() and passthru()**.

3.2.6 File Inclusion

Programs in PHP can be included in other program files through function **include**, **include_once**, **require**, and **require_once**. Tools need to claim malicious file inclusion may occur when files include external files. An example is in Figure 3.9:

3.2.7 Information Leakage

Analysis tools have to claim information leakage occurs when such sensitive function as phpinfo() or getMessage() are used, but attackers are expected to collect information through these error pages used in development phase. It is an example in Figure 3.10:

```
01: <?php
02:   $fileName = $_GET[filename];
03:   echo "<a href=$fileName.txt>Click here to see the file</a>";
04:   $file = file_get_contents($fileName, FILE_USE_INCLUDE_PATH);
05:
06:   $dirName = $_GET['dirN'];
07:   rmdir($dirName);
08:   echo $file;
09: ?>
```

Figure 3.6: File operation A.

```
01: <?php
02:   $name =$_POST['aName'];
03:   $number =$_POST['aNumber'];
04:   // file container where all texts are to be written
05:   $fileContainer = date("MjY").$name.'.log';
06:   // open the said file
07:   $filePointer = fopen($fileContainer,"w+");
08:   // text to be written in the file
09:   $logMsg = "Number is: ".$number
10:   // below is where the log message has been written to a file.
11:   fputs($filePointer,$logMsg);
12:   // close the open said file after writing the text
13:   fclose($filePointer);
14: ?>
```

Figure 3.7: File operation B.

```
01: <?php
02:   $statement = $_GET['statement'];
03:   exec($statement);
04:   passthru($statement);
05: ?>
```

Figure 3.8: Command execution.

```
01: <?php
02:   include 'includevar.php';
03:   //include_once 'includevar.php';
04:   //require 'requirevar.php';
05:   //require_once 'requirevar.php';
06:   $link = mysql_pconnect("localhost", "root", "svvrl");
07:   mysql_select_db("tabitha") or die("無法選擇資料庫");
08:   //二、執行SQL語法
09:   // 建立SQL語法
10:   $query = "SELECT * FROM stuInfo where department = '" .$IMdepartment. "'";
11:   $result = mysql_query($query) or die("無法送出" . mysql_error( ));
12:   while ( $row = mysql_fetch_array($result) ) {
13:     echo $row['stu_name']."<BR>";
14:   }
15:   mysql_free_result($result);
16: ?>
```

Figure 3.9: File inclusion.

```
01: <?php
02:   function inverse($x)
03:   {
04:     if (!$x) {
05:       throw new Exception('Division by zero.');
06:     }
07:   else return 1/$x;
08:   }
09:
10:   try {
11:     echo inverse(5) . "<br/>";
12:     echo inverse(0) . "<br/>";
13:   }
14:   catch (Exception $e) {
15:     echo 'Caught exception: '.$e->getMessage();
16:   }
17: ?>
```

Figure 3.10: Information leakage.

Chapter 4

Implementation and Evaluation

4.1 Implementation Overview

4.1.1 System Environment

In this chapter, we conduct the experiments with database software, MySQL 5.0.24a, and PHP version 5.1.6 to run program files (our test suite). All of experiments will be performed on an Hewlett-Packard PC with Intel Core 2 Quad CPU Q6600 @ 2.40GHz, 1.95GB main memory, run on Microsoft Windows Server 2003 Standard Edition Service Pack 2.

To refer to [34, 7], we selected four tools to implement experiences with benchmark cases, but names of tools are anonymous with "Tool A", "Tool B", "Tool C" and "Tool D". Except for Tool C, which claims it can test whether program is vulnerable in cross-site script or SQL injection only, others do not claim their limitations in vulnerability categories.

4.1.2 Vulnerability Categories

We divide Web security vulnerabilities into five categories:

1. Cross-Site scripting,
2. SQL injection,
3. Resource injection,
4. Dangerous functions and files, and

5. Information leakage.

Except the former three categories, we combine finer categories into the later two categories. Specifically, we merge command injection, PHP file inclusion, eval injection, reflection injection and variable overwrite into the fourth category **dangerous functions and files**; Information leak, error message information leak and information leak through debug information are combined into the fifth category **information leakage**.

4.1.3 Statistics Formulae

In this thesis, we use nine statistics formulae to show experimental result in each static analysis tool with four figures: true positive (It is a vulnerability, and the tool claims it is.), true negative (It is not a vulnerability, and the tool does not claim it is.), false positive (the tool reports bugs that program does not contain.), and false negative (the program contains bugs that tool does not report), counted in our experiments. Simplificatively, TN, TN, FP, and FN are abbreviated from these four nouns, respectively.

1. **Sensitivity** = TP / (TP + FN)
2. **Specificity** = TN / (TN + FP)
3. **Positive predict value** = TP / (TP + FP)
4. **Negative predict value** = TN / (TN + FN)
5. **False positive rate** = FP / (FP + TN)
6. **False negative rate** = FN / (FN + TP)
7. **False discovery rate** = FP / (FP + TP)
8. **False omission rate** = FN / (FN + TN)
9. **Correctness rate** = (TP + TN) / (TP + TN + FP + FN)

4.2 Evaluation by Categories

The more proportion of positive predict value and more negative predict value are, the better analysis tools are. On the other hand, the more ratio of false positive rate and more false negative rate, the worse analysis tools are. We display which parts are not enough in four targeted static analysis tools so that get rise to false negatives and false positives in this section.

4.2.1 Cross-Site Scripting

Items	**Tool A**	**Tool B**	**Tool C**	**Tool D**
Sensitivity	33.93%	69.64%	31.58%	78.57%
Specificity	81.82%	81.82%	81.82%	61.36%
Positive Predict Value	70.37%	82.98%	60.00%	72.13%
Negative Predict Value	49.32%	67.92%	58.06%	69.23%
False Positive Rate	**18.18%**	**18.18%**	**18.18%**	38.64%
False Negative Rate	66.07%	30.36%	68.42%	**21.43%**
False Discovery Rate	29.63%	17.02%	40.00%	27.87%
False Omission Rate	50.68%	32.08%	41.94%	30.77%
Correctness Rate	55.00%	75.00%	58.54%	71.00%

Table 4.1: Experimental result in cross-site scripting.

False negatives often appear in cross-site scripting when output values provided by users through function **print_r()** and **die()** ,or furnished by contents of result of database manipulation and contents of files. Although function **echo** is a common printing function, others ought to be included in the scanning scope. Sometimes attackers insert malicious values into database or files, so we need to validate these data before entering sensitive functions. Moreover, benchmark cases with variable variable and string index are not handled completely.

False positives frequently occur in cross-site scripting when the attribute **value** of a hidden type through function **echo**, but our experiment result shows attackers cannot execute scripts via, for instance, the following statement:

```
<input type='hidden' name='Hvalue' value='<?php echo $_GET['a']; ?>'>
```

Furthermore, static analysis tools should trace values of variables for determining if parameters of sinks is trustful. For instance, constant strings have to be viewed as untainted data.

4.2.2 SQL Injection

Items	Tool A	Tool B	Tool C	Tool D
Sensitivity	44.44%	55.56%	88.89%	100.00%
Specificity	100.00%	100.00%	98.51%	85.37%
Positive Predict Value	100.00%	100.00%	88.89%	42.86%
Negative Predict Value	94.25%	95.35%	98.51%	100.00%
False Positive Rate	**0.00%**	**0.00%**	1.49%	14.63%
False Negative Rate	55.56%	44.44%	11.11%	**0.00%**
False Discovery Rate	0.00%	0.00%	11.11%	57.14%
False Omission Rate	5.75%	4.65%	1.49%	0.00%
Correctness Rate	94.51%	95.60%	97.37%	86.81%

Table 4.2: Experimental result in SQL injection.

False negatives constantly emerge when a part of an SQL statement supplied by result of database manipulation and external files included into the current page, and tools ought to warn programmers it is dangerous to take an external component as a part of a database query.

Tools should suggest that sanitization functions are poor when developers use character-escaping functions, such as **mysql_real_escape_string()** in an SQL statement.

4.2.3 Resource Injection

False positives do not take place in benchmark cases we design, but false negatives appear when functions of file manipulation are used. File-handling functions (e.g., touch() and fread()), file-storing functions (e.g., error_log and fputs()), and a dynamic hyperlink-creating are ignored in our benchmark cases in resource injection. However, tools have to point out the program line where internal components are operated.

Items	Tool A	Tool B	Tool D
Sensitivity	80.00%	66.67%	26.67%
Specificity	100.00%	100.00%	100.00%
Positive Predict Value	100.00%	100.00%	100.00%
Negative Predict Value	96.30%	93.98%	87.64%
False Positive Rate	**0.00%**	**0.00%**	**0.00%**
False Negative Rate	**20.00%**	33.33%	73.33%
False Discovery Rate	0.00%	0.00%	0.00%
False Omission Rate	3.70%	6.02%	12.36%
Correctness Rate	96.77%	94.62%	88.17%

Table 4.3: Experimental result in resource injection.

4.2.4 Dangerous Functions and Files

Items	Tool A	Tool B	Tool D
Sensitivity	53.85%	61.54%	100.00%
Specificity	100.00%	100.00%	100.00%
Positive Predict Value	100.00%	100.00%	100.00%
Negative Predict Value	92.86%	93.98%	100.00%
False Positive Rate	**0.00%**	**0.00%**	**0.00%**
False Negative Rate	46.15%	38.46%	**0.00%**
False Discovery Rate	0.00%	0.00%	0.00%
False Omission Rate	7.14%	6.02%	0.00%
Correctness Rate	93.41%	94.51%	100.00%

Table 4.4: Experimental result in dangerous functions and files.

False positives do not take place in benchmark cases we design, but false negatives occur when procedures invoke functions or load files without limiting names of functions and files may containing malicious contents. An example is as follows, but tools must notify it is dangerous without placing restrictions on names of a file before used:

```
01: <?php
02:   $first="User";
03:   $str = $_SERVER['QUERY_STRING'];
04:   parse_str($str);
05:   echo $first;
06: ?>
```

4.2.5 Information Leakage

Items	Tool A	Tool B	Tool D
Sensitivity	22.22%	33.33%	33.33%
Specificity	100.00%	98.78%	100.00%
Positive Predict Value	100.00%	75.00%	100.00%
Negative Predict Value	92.13%	93.10%	93.18%
False Positive Rate	**0.00%**	1.22%	**0.00%**
False Negative Rate	77.78%	**66.67%**	**66.67%**
False Discovery Rate	0.00%	25.00%	0.00%
False Omission Rate	7.87%	6.90%	6.82%
Correctness Rate	92.31%	92.31%	93.41%

Table 4.5: Experimental result in information leakage.

False negatives usually emerge when such error-reminding functions are used as **debug_print_backtrace()**, **phpinfo()**, and **getMessage()**. It is an example as follows, but this function leaks not only a trace but also function names and program locations:

```
01:   ...
02:   echo "The back trace is ".debug_print_backtrace();
03:   ...
```

4.3 Summary

The correctness shows at Table 4.6, and these calculated figures are associated with benchmark cases. In cross-site scripting, a false negative often occurs in two particular functions: **print_r()** and **die()**; Within SQL injection, it often occurs if some part of an SQL statement got from external files or the previous result of database manipulation; In resource injection, it happens in a particular function: **touch()** and the value of a hyperlink provided by user inputs; Within dangerous functions and files, a false negative takes place , notably, in file inclusion or function invocation; In information leakage, sensitive information will be divulged through function **getMessage()** and **mysql_error()**, but tools should claim these functions are dangerous.

In addition to important security vulnerabilities, verified by static analysis tools so that developers of Web applications are expected to modify vulnerable code, tools should claim vulnerabilities those do not give rise to critical problems in Web applications, such as **hard-coded password**, **cookie not sent with SSL** and so on.

Instead of comparing which tool gets the perfect performance in verifying vulnerabilities in Web applications, the main point of testing four tools with benchmark cases we designed is to know which part of available tools is not good enough, even not enough, so that not only improve our research on developing analysis methods by which code can be viewed as vulnerable or not precisely and correctly, but also advance software industry providing analysis tools with valuable suggestion.

Items	**Tool A**	**Tool B**	**Tool C**	**Tool D**
Cross-Site Scripting	55.00%	**75.00%**	58.54%	71.00%
SQL Injection	94.51%	**95.60%**	97.37%	86.81%
Resource Injection	**96.77%**	94.62%		88.17%
Dangerous Functions and Files	89.01%	90.11%		**100%**
Information Leakage	92.31%	92.31%		**93.41%**

Table 4.6: Experimental result in correctness rate.

Chapter 5

Methods Tested

5.1 Static Analysis Methods

Verification tools have implemented analysis methods from software applications [28] to Web applications for a long time, and an overview of static analysis methods applied to security vulnerabilities can be found in [4]. Collectively, static analysis methods are performed by pattern-matching [44], pointer-analyzing, type-checking. Pattern-matching is also know lexical analysis, and the methods look for syntax matching rules. Although more advanced tools allow new rules being added immediately, they need human efforts. An analysis method using type qualifier successfully solves format string and user/kernel bugs in C language [16, 36]. JFlow [27] also takes type-checking to verify Java code with information-flow analysis, but it is time-consuming to annotate types by human. However, it reserves an open question in verifying type-free scripting language such as PHP. Huang et al. [13, 14] perhaps first address SQL injection in Web applications in PHP, and we will describe their approach in the section 5.1.1. WebSARRI sees a sensitive sink being untaint if an untrustful source has been operated with sanitization functions, but false positive occurs if it does not track values of strings. Pixy [17, 18] determines whether a variable is vulnerable by using alias analysis to record the value of a string at each program location, and we illustrate the algorithm of Pixy in the section 5.1.2.

5.1.1 WebSSARI

WebSSARI [13, 14] is a verifier for Web application dealing with PHP, statically verifying program code at development period using typestate-based polynominal-time algorithm

(TS), in the former version, and bounded model checking (BMC), in the later version, respectively. Moreover, runtime guards are inserted at program sites where violation of safety policies is and are used to sanitize arguments before transferring arguments as inputs of sensitive functions.

System Overview

The WebSSARI system at development stage uses two different algorithms to verify program code of Web application, and the later algorithm improves on the efficiency of former one:

* **Typestate-based polynominal-time approach.** The approach considers vulnerabilities while a system takes tainted data provided from external components as arguments used in sensitive functions. However, it keeps type information by *type judgment rules* to check whether the program code is vulnerable.

* **Bounded model checking.** Since typestate-based polynominal-time approach is used for finding arguments checked with type judgment rules, it figures out lots of vulnerable program points which would be handled with sanitization functions by programmers. However, bounded model checking introduces counterexamples to group vulnerable program sites and to find sources of exposed program variables, and then programmers deploy sanitization functions efficiently.

Typestate-Based Polynominal-Time Approach

Some functions must be called with trusted arguments, such as **exec()** executing commands and **echo()** generating outputs, and we call these functions as **sensitive functions**. Putting preconditions in front of sensitive functions is an intuitive way to guard and guarantee arguments are benign.

Information Flow Model To build levels of trusted data, the WebSSARI system follows Denning's lattice model [8] and makes assumptions as follows:

1. Each variable is distributed a security class, a trust level.

2. T = $\{\tau_1, \tau_2, ..., \tau_n\}$ is a finite set of trust levels.

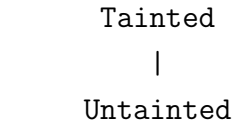

Figure 5.1: Primitive lattice.

3. T is partially ordered by $\leq$. For τ_1, τ_2, $\tau_3 \in$ T,

   $\tau_1 \leq \tau_1$ (reflexive),

   if $\tau_1 \leq \tau_2$ and $\tau_2 \leq \tau_1$ then $\tau_1 = \tau_2$ (anti-symmetry),

   if $\tau_1 \leq \tau_2$ and $\tau_2 \leq \tau_3$ then $\tau_1 \leq \tau_3$ (transitivity).

4. T forms a complete lattice with a lower bound and an upper bound, that is,

   $\forall\tau \in$T, $\perp \leq \tau$,

   $\forall\tau \in$T, $\tau \leq \top$.

Following Foster et al. [9] and Shankar et al. [36], the system extends the PHP language with extra *type qualifiers*, an annotation mechanism for expression types, explicitly associates security levels with variables and functions in programs, and assists the type qualifiers with three preludes: preconditions to verify inputs before sensitive function starting up, postconditions generating sanitized outputs from tainted inputs, and potential tainted input providers (e.g., $_GET, $_POST, $_REQUEST.)

According to Storm and Yemini's typestate [38] to change variable classes and to consider the flow-sensitive properties, a type environment $\Gamma : X \mapsto T$, a mapping function, maps from variables to security levels at a particular program site. For each variable $x \in dom(\Gamma)$, $\Gamma(x)$ is denoted mapped a security type τ of $x \in X$. At runtime, the security type of variables is regarded as a *static most restrictive class* of variables at each program point. For example, (assume t_1 is a tainted variable and u_1 is an untainted one) with widely adopted the tainted-untainted (T-U) lattice of security levels (Figure 5.1), the sensitive function **exec()** at line 2 of Table 5.2 requires an untainted argument. Since x can be either tainted(t_1) or untainted(u_1), $\Gamma(x) = tainted \sqcup untainted$.

Type-Aware Security Classes As all HTTP variables are stored as string format, regardless their real types, a single cast to sanitize tainted inputs is a common way. For

```
01: if(condition) x = t1; else x = u1;
02: exec(x);
```

Figure 5.2: Example A: exec().

```
01: $i = (int) $_POST['index'];
02: $s = (string) $i;
03: echo "<hidden name = mid, value = '$s'>"
```

Figure 5.3: Example of a false positive resulting from type-casting.

instance, in Table 5.3 programmers view variable $i as tainted data after line 1, and variable $s is regarded as a tainted input after line 2 since $_POST is a tainted input provider. However, the sensitive function **echo()** can accept tainted integers without compromising system integrity. Therefore, the system replace the tainted-untainted lattice with the type-aware one (Figure 5.4).

Program Abstraction and Type Judgment Firstly, the system filters program to abstract itself with the following syntax rules:

(*commands*) $c ::= c_1; c_2 \mid x := e \mid e \mid$ if e than c_1 else c_2

(*expressions*) $c ::= x \mid n \mid e_1 \sim e_2 \mid f(x)$

,where x is a variable, n is a constant value, $\sim$ is a binary operator (e.g., +), and *f(x)* is a function call. Given a program P and its initial type environment Γ_0, and then the validity of P depends on whether we can derive the judgment $\Gamma_0 \vdash P \rightarrow \Gamma$ with the judgment rules as Figure 5.5.

Bounded Model Checking

To guarantee outputs are untainted after executing sensitive functions, user inputs, usually viewed as tainted data, are sanitized by sanitization functions. The reason why the WebSSARI improves the algorithm from typestate-based polynominal-time algorithm to bounded model checking (BMC) is that BMC provides counterexamples allowing programmers patching at particular program sites where vulnerabilities are originally introduced instead of propagated ones.

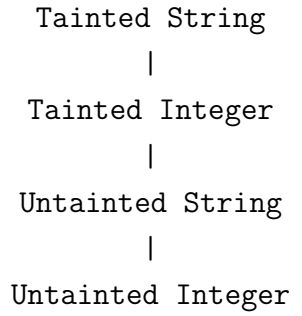

Figure 5.4: Type-aware lattice.

$$\frac{f \in T}{\Gamma \vdash f(x) \to \Gamma[x \mapsto \mathit{tainted}]} k(\text{Polution}) \qquad \frac{f \in S}{\Gamma \vdash f(x) \to \Gamma[x \mapsto \mathit{untainted}]} (\text{Sanitation})$$

$$\Gamma \vdash x := e \to \Gamma[x \mapsto \Gamma(e)] (\text{Assignment}) \qquad \frac{\Gamma \vdash C_1 \to \Gamma_1 \quad \Gamma \vdash C_2 \to \Gamma_2}{\Gamma \vdash \text{if } e \text{ then } C_1 \text{ else } C_2 \to \Gamma_1 \oplus \Gamma_2} (\text{Restriction})$$

$$\frac{f \in C \quad \mathit{satisfy}(\Gamma, f, x)}{\Gamma \vdash f(x) \to \Gamma} (\text{Precondition}) \qquad \text{, where } \mathit{satisfy}(\Gamma,\ f,\ x) \text{ checks if } \Gamma(x) \text{ satisfies the precondition of } f$$

$$\frac{\Gamma \vdash C_1 \to \Gamma_1 \quad \Gamma_1 \vdash C_2 \to \Gamma_2}{\Gamma \vdash C_1; C_2 \to \Gamma_2} (\text{Concatenation})$$

$$\Gamma(n) = \mathit{untainted} \quad \Gamma(e_1 \sim e_2) = \Gamma(e_1) \sqcup \Gamma(e_2) \text{ (Mapping Rules)}$$

Figure 5.5: The judgment rules.

Program Translation With bounded model checking, the system translates program code into Conjunctive Normal Form (CNF) formulae through four stages, and then uses efficient SAT solver zChaff [26] to check if counterexamples exist.

Firstly, in program filtering, given a program P, the system uses a filter to generate *F(P)* only containing assignments, function calls, and conditional structures with following syntax rules:

$c ::= x{:=}e \mid f_i(X) \mid f_o(X) \mid \text{stop} \mid \text{if } e \text{ then } c_1 \text{ else } c_2 \mid \text{while } e \text{ do } c \mid c_1;c_2$

$e ::= x \mid n \mid e_1 \sim e_2$

,where x is a variable ($\forall_{x \in X} \subseteq \text{dom}(P)$), n is a constant value, $\sim$ is a binary operator (e.g., +), $f_i(X)$ is denoted as an *untrusted input channel* (e.g., GET_HTTP_VARS), and $f_o(X)$ is identified as a *sensitive output channel* requiring untainted data (e.g., echo()).

Secondly, the system translates *F(P)* into abstract interpretation (AI) consisting of instructions, type assignments, and assertions as Figure 5.6. An *assignment* from ex-

pression e to variable x is denoted that propagating security type of e to x; Function *postcondition* are set with security type ($\forall_{x \in X} t_x = \tau$, $\tau \in \mathrm{dom}(T)$) of retrieved data; Function *precondition* asserts security types to parameters of sensitive functions, and ($assert(\forall_{x \in X} t_x < \tau)$ means every variable x in X is safer than security type τ).

Filtered Result: $F(p)$	**Abstract Interpretation: $AI(F(p))$**
$x = e$	$t_x = t_e$, where $t_n = \bot$, $t_{e_i \sim e_j} = t_{e_i} \sqcap t_{e_j}$
$f_i(X)$	$\forall_{x \in X} t_x = \tau$, where τ depends on the postcondition of f_i
$f_o(X)$	$assert(\forall_{x \in X} t_x < \tau)$, where τ depends on the precondition of f_o
stop	stop
if e then c_1 else c_2	if b_e then $AI(c_1)$ else $AI(c_2)$
while e do c	if b_e then $AI(c)$
c_1;c_2	$AI(c_1)$; $AI(c_2)$

Figure 5.6: The translation from filtered result to abstract interpretation.

Thirdly, the purposes of renaming are removing duplicate assignments [5] and indicating the specificate program site where tainted variables are.

Finally, given a command c, the constraints C(c, g) are introduced with rules as Figure 5.7 (g is a guard and initially is true):

***AI*(Command)**	**Constraint**
$t_x = t_e$	$C(c, g) := {t^i}_x = g \ ? \ \rho(t_e) : {t^{i-1}}_x$
$assert(\forall_{x \in \ X} t_x < \tau)$	$C(c, g) := g \Rightarrow \rho\big(\bigcap_{x \in X} t_x < \tau\big)$
stop	$C(c, g) := true$
if b_e then c_1 else c_2	$C(c, g) := C(c_1, g \wedge b_e) \wedge C(c_2, g \wedge \neg b_e)$
c_1;c_2	$C(c, g) := C(c_1, g) \wedge C(c_2, g)$

Figure 5.7: The translation from abstract interpretation to constraint.

The system checks one assertion ($assert_i$) at a time, and each assertion concatenates preceding programs p as $p; assert_i$, generating counterexamples until a CNF formula is unsatisfied, whenever the system collects all counterexamples. Figure 5.8 is an example of whole translation processes (assume the postcondition of the function $_GET is tainted and function **htmlspecialchars** is untainted.)

PHP Code	**Filter**	**Abstract Interpretation**

...
if(*condition*) {
$tmp = $_GET["user"];
echo(
htmlspecialchars($tmp)
);
} else {
$temp = "You are the ".
$GuestCount." guest";
echo($temp);
}
...
	...
	if(*condition*) {
	$f^t{}_i(user)$;
	$tmp = user$;
	$f^u{}_i(tmp)$;
	$f_o(tmp)$;
	} else {
	$tmp = GuestCount$;
	$f_o(tmp)$;
	}
	...
		...
		if $b_{condition}$ then
		$t_{user} = tainted$;
		$t_{tmp} = t_{user}$;
		$t_{tmp} = untainted$;
		$assert(t_{tmp} < tainted)$;
		else
		$t_{tmp} = t_{GuestCount}$;
		$assert(t_{tmp} < tainted)$;
		...

Renaming	**Constraint for Each Assertion**

...
if $b_{condition}$ then
$t^i{}_{user} = tainted$;
$t^j{}_{tmp} = t^i{}_{user}$;
$t^{j+1}{}_{tmp} = untainted$;
$assert_k(t^{j+1}{}_{tmp} < tainted)$;
else
$t^{j+2}{}_{tmp} = t^k{}_{GuestCount}$;
$assert_{k+1}(t^{j+2}{}_{tmp} < tainted)$;
...
	$B_k := (t^i{}_{user} = b_{condition} \ ?\ tainted : t^{i-1}{}_{user}) \wedge$
	$(t^j{}_{tmp} = b_{condition} \ ?\ t^i{}_{user} : t^{j-1}{}_{tmp}) \wedge$
	$(t^{j+1}{}_{tmp} = b_{condition} \ ?\ untainted : t^j{}_{tmp}) \wedge$
	$\neg(b_{condition} \Rightarrow t^{j+1}{}_{tmp} < tainted)$
	$B_{k+1} := (t^i{}_{user} = b_{condition} \ ?\ tainted : t^{i-1}{}_{user}) \wedge$
	$(t^j{}_{tmp} = b_{condition} \ ?\ t^i{}_{user} : t^{j-1}{}_{tmp}) \wedge$
	$(t^{j+1}{}_{tmp} = b_{condition} \ ?\ untainted : t^j{}_{tmp}) \wedge$
	$(b_{condition} \Rightarrow t^{j+1}{}_{tmp} < tainted) \wedge$
	$(t^{j+2}{}_{tmp} = \neg b_{condition} ?\ t^k{}_{GuestCount} : t^{j+1}{}_{tmp}) \wedge$
	$\neg(\neg\, b_{condition} \Rightarrow t^{j+2}{}_{tmp} < tainted)$

Figure 5.8: An example of whole translation processes from PHP code to Constraint.

Efficient Sanitization The WebSSARI system provides an efficient way to sanitize vulnerable variables. Let v be an individual violating variable, V be a set of violating variables, r be an individual error trace, and R be a set of error traces. The definition of an effective sanitization function is in the following statement:

Definition 1 (The effectiveness of sanitization)**.** *A Fix*(V)*, after sanitizing all variables* $v \in V$*, is an effective function, for the error trace* r*.*

With a minimal fixing set V_r for every error trace $r \in R$, Fix(V_R) serves as an effective fix function for each error trace r. However, V_R is not a minimum set of violating variables.

To find the minimum set of violating variables, for every violating variable $v_\alpha \in V_r$, a set S_{v_α} is built from adding variables along an error trace r. That it,

$$s_{v_\alpha} = \begin{cases} \{v_\alpha\} \cup s_{v_\beta}, & \text{if the form of the single assignments is } v_\alpha = v_\beta \\ \{v_\alpha\}, & \text{otherwise.} \end{cases}$$

Because each set of violating variable S_{v_α} along an error trace r contains propagated variables introduced from the original program site, sanitizing any variable in a set of violating variables is the same as sanitizing the violating variable v_α. In other words, if a fixing function $\text{Fix}(V_r)$, for any $v_\alpha \in V_r$, is an effective function for an error trace r, the fix function $\text{Fix}(V_r[v_\beta/v_\alpha])$ is an effective fix function as well, for any $v_\beta \in S_{v_\alpha}$. Finally, if the intersection between the minimum effective fixing set ${V^m}_R$, $\forall v \in V$ and $\forall r \in R$, and the set of violating variables S_{v_α} is non-empty, the fix function $\text{Fix}({V^m}_R)$ is an efficient fix function. Therefore, developers fix every variable in minimum fixing set is an efficient way to sanitize vulnerable variables.

5.1.2 Pixy System

Pixy [17, 18] is a static analysis tool for security vulnerabilities containing cross-site scripting and SQL injection in PHP scripting language. The semantic of references in PHP practically differs from pointers in traditional languages such as C and Java [43]. References in PHP, that can change data types of variables at each program site, are symbol table alias [31] instead of memory locations, while C pointers are particular direction to memory addresses. Although [18] mentions the algorithm checks find cross-site scripting, the practical system is expected to check cross-site scripting and SQL injection.

Alias Analysis in PHP

Two or more variables are *alias* when their values at specific program line are stored in the same memory location. In the Pixy system, two variables belong to *must-alias* group if they are alias no matter what their actual execution paths are at run time, and two or more variables are a member of *may-alias* group if they are alias only at some program paths.

Compared with pointers in traditional program such C, where semantics of formal parameters effect that of actual parameters, the *reference operator* **$** in PHP for aliases of variables applies without deviation in an assignment, or in combination with actual parameters and formal parameters. An example is given in Figure 5.9. The output of variable x1 is still '1' at line 4 regardless of the actual parameter x1 at line 3 and the formal parameter $\$p$ at line 6 are alias due to the parameter, passing and then $\$p$ and x2 at line 7 are alias.

```
01: $x1 = 1;
02: $x2 = 2;
03: a(&$x1);
04: echo $x1;                          // $ x1 is still '1'
05:
06: function a(&$p) {
07:     $p = &$GLOBALS['x2'];
08: }
```

Figure 5.9: Semantic references in contrast with pointers.

Moreover, alias analysis emphasizes the quality and quantity of vulnerability reports. In other words, false alarms (false positives and false negatives) can be caused by lacking information of alias analysis. In Figure 5.10 showing an example of a false positive, variable $x3 aliases with $x4 at line 2, and we think $x4 at line 4 is untainted without alias analysis information. Analogously, a false negative is illustrated by Figure 5.11. Variable $x5 and $x6 are alias at line 2, and then $x5 at line 4 is considered as tainted output.

```
01: $x4 = 'nice';          // $x4: untainted
02: $x3 = $x4;             // $x3 and $x4: untainted
03: $x3 = $GET['evil']     // $x3 and $x4: tainted
04: echo $x4;              // without alias analysis,
                              $x4 would be considered untainted
```

Figure 5.10: False positive without alias analysis

```
01: $x5 = $GET['evil'];    // $x5: tainted
02: $x5 = &$x6;            // $x5 and $x6 are aliased
03: $x6 = 'nice';          // $x5 and $x6: untainted
04: echo $x5;              // without alias analysis,
                              $x5 would be considered malicious
```

Figure 5.11: False negative without alias analysis

Intraprocedural Alias Analysis

In Figure 5.12, we present *must-alias group* with u and *may-alias group* with a, disjoining elements in must-alias group unorderedly. At line 2, the first element in must-alias group is the pair of $x7 and $x8, since $x7 and $x8 are aliased in any program path. The second element in must-alias group in the pair of variable $x9 and $x10 at line 4, because$x9 directs to $x10; the second element extends $x11 as result of the statement at line 5. However, we separate ($x9, $x10, $x11) into ($x9, $x10), ($x10, $x11), and ($x9, $x11) outside the *if* statement to perform intraprocedural analysis.

The analysis separates *must-alias group* from *may-alias group* to improve the accuracy of vulnerability reports. For example, variable $y is tainted if a pair of variable $x and

```
01: skip;                 // u{} a{}
02: $x7 = &$x8;           // u{($x7, $x8)} a{}
03: if (condition) {
04: $x9 = &$x10;          // u{($x7, $x8) ($x9, $x10)} a{}
05: $x11 = &$x10;         // u{($x7, $x8) ($x9, $x10, $x11)} a{}
06: }
07: skip;                 // u{($x7, $x8)} a{($x9, $x10), ($x10, $x11), ($x9, $x11)}
```

Figure 5.12: Intraprocedural alias analysis

$y is an element in must-alias group, so that developers have to sanitize $y, an input of a sensitive function, whenever the system detects that $x is tainted.

Interprocedural Alias Analysis

Interprocedural alias analysis has to handle the recursive function calls. How deep the loop depends on external resources, such as values originally transformed from a database or user inputs, and it may be undetermined and infinite. To solve this problem is to limit the number of targeted subjects, and two principles shows in the following description:

1. In a function (local) scope, the analysis only checks information about global variables and local variables inside a function.

2. In a global scope, the analysis only tracks global variables.

The system considers many cases according to different program operations, such as parameter passing, via two perspectives: viewpoints of a callee and a caller, namely.

From a standpoint of a callee, we consider the following information:

- Aliasing among global variables.

- Aliasing among formal parameters of a callee.

- Aliasing between global variables and formal parameters of a callee.

From a point of a caller, we contemplate of following information after a callee returns values:

- Aliasing among global variables.
- Aliasing between global variables and local variables of a caller.

Aliasing among Global Variables Both of a caller and a callee consider issues about global variables. On one part, callees have to know how global variables are aliased when a function call is executed. On the other part, a caller needs to know what value of global variables modified by callees is. An example is demonstrated in the Figure 5.13. We identify names of functions with prefixions, for example, prefixion **m** presents function **main** containing global variables. Function **b** is performed at line 2, and the analysis applies intraprocedural aliasing until program site is at line 9. However, before entering the function **c**, elements in must-alias group except local variables of a caller (function **b**) are propagated into function **c**. After redirecting global variable $x14 to $x12 at line 15, the elements containing $x12 and $x13 extends $x14 as result of aliasing among global variables. At line 10, with returning to function **b**, must-alias group restores local variables of a callee. Similarly, aliasing in may-alias group are treated analogously.

```
01: skip;                                          // u{} a{}
02: b();
03: skip;                                          // u{(m.x12, m.x13, m.x14)} a{}
04:
05: function b() {                                 // u{} a{}
06:     $b1 = &$b2;                                // u{(b.b1, b.b2)} a{}
07:     $GLOBALS['x12'] =& $GLOBALS['x13'];
08:     skip;                                      // u{(b.b1, b.b2) (m.x12, m.x13)} a{}
09:     c();
10:     skip;                                      // u{(b.b1, b.b2)
11:                                                //   (m.x12, m.x13, m.x14)} a{}
12: }
13:
14: function c() {                                 // u{(m.x12, m.x13)} a{}
15:     $GLOBALS['x14'] = &$GLOBALS['x12'];
16:     skip;                                      // u{(m.x12, m.x13, m.x14)} a{}
17: }
```

Figure 5.13: Aliases between global variables.

Aliasing among Formal Parameters of a Callee Aliasing among formal parameters exists while aliasing relationship occurs among the corresponding actual parameters. In Figure 5.14 offering an example, since formal call-by-value parameters $**ep1** and $**ep2** at line 8 correspond to actual call-by-value parameters $**d1** and $**d2** at line 5, must-alias group at line 9 is delivered by that at line 4 before an entry of function **e**.

```
01: d();
02:
03: function d() {                    // u{} a{}
04:     $d1 = &$d2;                   // u{(d.d1, d.d2)} a{}
05:     e(&$l1, &$l2);
06: }
07:
08: function e(&$ep1, &$ep2) {
09:     skip;                         // u{e.ep1, e.ep2)} a{}
10: }
```

Figure 5.14: Must-Aliases among formal parameters.

With may-alias group in *alias among formal parameters*, Table 5.1 provides all possible cases and elements in a may-alias group. Firstly, the number of actual call-by-value parameters definitely discriminates between one and two (in the first column of Table 5.1). Secondly, with unorderly features of a may-alias group, may-alias pairs are distinguished from three types: (local, local), (global, global), and (local, global) (in the second column of Table 5.1). We give you a specific example between the second row of Table 5.1 and code in Figure 5.15. The may-alias pair ($f1, $f2) reaches the function **g** at line 8. On the entry of function **g**, the analysis initially creates three may-alias pairs: ($gp1, $gp2), ($gp1, $f2), and ($f1, $gp2), and then deletes the later two pairs since local variables cannot be transformed into a callee.

Aliasing between Global Variables and Formal Parameters of a Callee We have to consider the following two cases in *alias between global variables and formal parameters of a callee*:

- The parameter is a must-alias of a global variable.

Function call	Entering may-aliases	Resulting relevant may-aliases	Resulting irrelevant may-aliases
f(&$l1, -)	(l1, l2)	none	(p1, l2)
f(&$l1, &$l2)	(l1, l2)	(p1, p2)	(p1, l2), (p2, l1)
f(&$g1, -)	(g1, g2)	(p1, g2)	none
f(&$g1, &$g2)	(g1, g2)	(p1, g2), (p2, g1), (p1, p2)	none
f(&$l, -)	(l, g)	(p1, g)	none
f(&$g, -)	(l, g)	none	(p1, l)
f(&$l, &$g)	(l, g)	(p1, p2), (p1, g)	(p2, l)

Table 5.1: May-Aliases among formal parameters resulting from function call execution.

- The parameter is a may-alias of a global variable.

Figure 5.16 gives an example of the first case (the parameter is a must-alias of a global variable). For calling function **i** at line 5, variable $h1 is a part of an element of a must-alias group, and $h1 aliases with global variable $x15. Since $h1 is an actual call-by-reference parameter, the formal call-by-reference parameter $ip1, entering function **i**, becomes a part of an element of a must-alias group.

Aliasing between Global Variables and Local Variables of a Caller The *alias among local variables of a caller* cannot be modified by callees, but relationship between local variables of a caller and global variables can be changed by callees. The system uses two shadow variables to aid classification between a must-alias set and a may-alias set: *formal-shadows* (for each formal parameter), introduced and aliased with its corresponding formal parameter at the beginning of a function, and *global-shadows* (for every global variable), created and aliased with its corresponding global variable at the start of a function. An example is demonstrated in the Figure 5.17. With a call of function **k** at line 9, local variable $j1 aliases with a global variable $x16, and then the procedure transforms must-alias group into function **k**. Without global-shadows, the analysis cannot know that $j1 does not alias with $x16 when $x16 redirects to a global variable $x17 at line 17. At line 10, with returning to function **j**, we can inferentially know that $j1 is not aliased with $x16 via a member of a must-alias set (j.j1, j.x16_gs).

```
01: f();
02:
03: function f() {                    // u{} a{}
04:     (condition) {
05:         $f1 = &$f2;               // u{f.f1, f.f2)} a{}
06: }
07:     skip;                         // u{} a{f.f1, f.f2)}
08:     g(&$f1, &$f2)
09: }
10:
11: function g(&$gp1, &$gp2) {
12:     skip;                         // u{} a{g.gp1, g.gp2)}
13: }
```

Figure 5.15: May-aliases among formal parameters.

5.1.3 Summary

Some tools point sensitive sinks their corresponding untrustful sources, but some only indicate the program location of sinks they claim. To fix these security vulnerabilities, developers design sanitization functions after an external component comes or before sensitive information is displayed to the public. In the other point of view, the effectiveness of sanitization functions is of importance. If tools regard sanitization functions as being effective, false negatives may occur. On the other hand, if tools consider that sanitization is ineffective, false positives may appear.

Since little research has been performed to determine if sanitization functions are effectual, we will illustrate the idea of Saner [2] composing static and dynamic analysis for validating sanitization functions in Web applications. Verifying the effectiveness of sanitization not only decreases false positives but also informs sanitization designers their sanitization functions are incorrect.

```
01: h();
02:
03: function h() {                    // u{} a{}
04:     $h1 = &$GLOBAL['x15'];        // u{(h.h1, m.x15)} a{}
05:     i(&$h1);
06: }
07:
08: function i(&$ip1) {
09:     skip;                         // u{m.x15, i.ip1)} a{}
10: }
```

Figure 5.16: Must-aliases between formal parameters and global variables.

5.2 Supplement Methods

Saner [2] combining static and dynamic techniques identifies sanitization routines where a malicious value may bypass. Developers sometimes do not think over every program path entirely so that give an attacker an opportunity to insert malicious inputs that can reach sensitive functions. Balzarotti et al. use two phases to implement supplement methods to verify the effectiveness of sanitization routines. Firstly, sanitization graph is generated for each pair of a sink and a source provided by the static analysis result. Secondly, benchmark cases are inserted into each path of a sanitization path.

5.2.1 Extracting the Sanitization Graph

For a given pair of a sink and a source, the sanitization graph is a fragment of interprocedural data-flow graph. Nodes in a fragment of interprocedural data-flow graph contain values exerting impacts on content of a sink, and the graph keeps paths passing through sanitization routines divided into language-provided sanitization routines (e.g., strip_tags and htmlspecialchars), regular-expression-based substitutions (e.g., preg_replace), string-based substitutions (e.g., str_preplace and stroupper), and other built-in string operations (e.g., concatenation). Figure 5.18 is an example program, and cross-site scripting occurs at line 12 if the value of $name propagated from external component at line 11 is not verified or even is checked incorrectly and incompletely. Before sanitization routines at line 5 or 7, the value will pass through the regular-expression-based substitution func-

```
01: j();
02: skip;                                          // u{(m.x16, m.x17)} a{}
03:
04: function j() {                                 // u{(m.x16, j.x16_gs)
05:                                                //   (m.x17, j.x17_gs)} a{}
06:    $j1 =& $GLOBALS['x16'];
07:    skip;                                       // u{(m.x16, j.x16_gs, j.j1)
08:                                                //   (m.x17, j.x17_gs)} a{}
09:    k();
10:    skip;                                       // u{(m.x16, m.x17, j.x17_gs)
11:                                                //   (j.j1, j.x16_gs)} a{}
12: }
13:
14: function k() {                                 // u{(m.x16, k.x16_gs)
15:                                                //   (m.x17, k.x17_gs)} a{}
16:
17:    $GLOBALS['x16'] =& $GLOBALS['x17'];
18:
19:    skip;                                       // u{m.x17, k.x17_gs, m.x16)} a{}
20: }
```

Figure 5.17: Aliases between global variables and the caller's local variables.

tions at line 3. Nodes associating with these three functions are connected together in a data-flow graph to show the sequence among these three routines, and its sanitization graph showed in the Figure 5.19 for a given pair of a sink and a source.

5.2.2 Testing the Effectiveness of Sanitization Routines

Since static analysis tools point a path between a source and a sink, we note the path with $\mathbf{P_i}$ and extract a block of code $\mathbf{C_i}$ that could be interpreted by PHP for each $\mathbf{P_i}$. This operation may need handling values to determine parameters used in a function call. For instance, the first and second parameter fetched in the function **eregi_replace** are string constants.

Balzarotti et al. select an appreciate test suite used in their experiment in accordance with different categories of sinks (cross-site scripting and SQL injection.) Both test suites

```
01: <?php
02:   functino sanitize($data) {
03:     $res = eregi_replace("<script", "", $data);
04:     if(version_compare(phpversion(), "4.3.0")=="-1")
05:       $res = mysql_escape_string($res);
06:     else
07:       $res = mysql_real_strin($res);
08:     return $res
09: }
10:
11: $name = sanitize($_GET['username']);
12: echo "Name: ". $name;
13: ?>
```

Figure 5.18: Customized sanitization function.

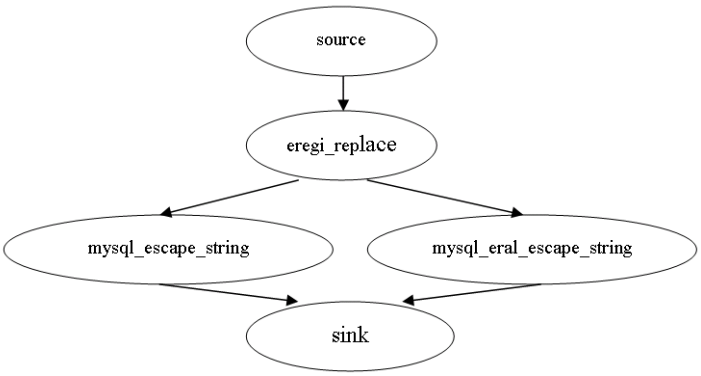

Figure 5.19: Sanitization graph.

contain a large number of test cases, and each of them includes attack strings that may successfully exploits vulnerabilities. These two test suites created by authors' experiments and from specialized Web sites [33, 24]. As examples, the following are three test cases to check the effectiveness of sanitization routines of cross-site scripting.

```
<script>alert(1);</script>
<scr<scriptipt src = http://evil.com/attack.js>
<img onmouseover = "alert(1);" src=""></img>
```

Finally, authors use the PHP interpreter to evaluate the result of executing each block of code $\mathbf{C_i}$ to determine if sanitization functions are evaded by their test cases.

Chapter 6

Conclusion

In this thesis, our objective is to evaluate the efficacy of static verification tools instead of dynamic analysis tools [22, 3, 11, 19, 29, 32, 15]. To this end, we build benchmark cases of vulnerable code that may cause security problems, such as cross-site scripting and SQL injection, but some benchmark cases do not consist of vulnerable code to determine if a false positive occurs after the tool scans the code. Specifically, we use the developed benchmark cases to test four static analysis tools that generate reports of vulnerable program sites, and evaluate the performance of the tools statistically. Moreover, the benchmark cases enable us to identify the structures or control flow statements that cause false alarms in the four tools. As a result, we can determine which benchmark cases are not handled in the target tools.

6.1 Contributions

1. *Build benchmark cases for evaluating the accuracy of existing static analysis tools*:

   To determine whether or not the result of scanning problem code is correct, we design a test suite to assess if the scanning result of static analysis is correct. Specifically, we determine 1) if the vulnerability claimed by static analysis tools is a false positive (the program is not vulnerable, but the tool claims it is) or a true positive (the program is vulnerable, and tool claims that it is); 2) manually check if the tools make wrong judgments about a false negative (the code is vulnerable, but tool does not recognize the problem.) Moreover, we add seldom used structures, such as variable variable and string index and control flow statements to determine

if available tools can handle more complicated data structures.

2. *Use the benchmark on the target tools to evaluate the precision of static analysis tools*:

   We integrate the result reported by four available static analysis tools to illustrate their scanning performance in different vulnerability categories based on criteria like the false positive rate, false negative rate and correctness rate. Furthermore, we show that a false alarm often occurs in certain functions or syntax of PHP scripting language, and aggregate the useful information for the later research.

3. *Produce useful feedbacks for tool designers*

   The benchmark cases enable us to identify the structures that cause false alarms in the four tools. As a result, we can determine which benchmark cases are not handled in the target tools (e.g., the structure of the variable variable and the string index), and then collect useful information for developers of static analysis tools.

6.2 Future Work

1. *Study more scripting languages and more analysis tools*:

   In this thesis, our target scripting language is PHP and the number of experiment tools is four. It is desirable to study other scripting languages, such as JSP and ASP.NET, as well as other analysis tools in our future work. Moreover, we need to incorporate more complex structures into our benchmark cases to determine whether the analysis tools can handle them.

2. *Build an application for implementing experiments, integrating the result of reports, and evaluating the performance of each targeted tool automatically*:

   Conducting experiments, integrating reports, and performance evaluations are currently handled manually. An automatic application not only speeds these processes but also implements more test cases to study which parts of the available analysis tools are inaccurate.

3. *Gather benchmark cases on the Web*:

   Since hackers continually generate new and advanced methods to attack Web applications, there is an urgent need to design benchmark cases to identify different security vulnerabilities in Web applications. It is desirable to implement benchmark cases for static analysis tools with various scripting languages. Thus, building a Web database to collect benchmark cases from Web contributors is worthwhile.

4. *Evaluate the qualification of the benchmark*:

   We have built 90 benchmark cases to evaluate the effectiveness of existing verification methods and tools, and it is desirable to establish measurements of benchmark cases in each vulnerability categories to evaluate the qualification of the benchmark.

Bibliography

[1] D. Baca, B. Carlsson, and L. Lundberg. Evaluating the cost reduction of static code analysis for software security. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, pages 79–88. ACM New York, NY, USA, 2008.

[2] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing static and dynamic analysis to validate sanitization in Web applications. In *Security and Privacy, IEEE Symposium on*, pages 387–401. IEEE Computer Society, 2008.

[3] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, and G. Vigna. SNOOZE: Toward a stateful network protocol fuzzer. In *9th Information Security Conference*, pages 343–358, 2006.

[4] B. Chess and G. McGraw. Static analysis for security. *IEEE Security and Privacy*, pages 76–79, 2004.

[5] E. Clarke, D. Kroening, and K. Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *Proceedings of the 40th Conference on Design Automation*, pages 368–371. ACM New York, NY, USA, 2003.

[6] F. Coelho. PHP-related vulnerabilities on the national vulnerability database. http://www.coelho.net/php_cve.html, 2009.

[7] CWE. Common weakness enumeration. http://cwe.mitre.org/, 2009.

[8] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, pages 236–243, 1976.

[9] J.S. Foster, M. Fähndrich, and A. Aiken. A theory of type qualiers. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 192–203. ACM New York, NY, USA, 1999.

[10] The PHP Group. Usage stats for April 2007. http://www.php.net/usage.php, 2009.

[11] V. Haldar, D. Chandra, and M. Franz. Dynamic taint propagation for Java. In *Proceedings of the 21st Annual Computer Security Applications Conference*, pages 303–311. IEEE Computer Society Washington, DC, USA, 2005.

[12] W.G.J. Halfond and A. Orso. AMNESIA: Analysis and monitoring for neutralizing SQL-injection attacks. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 174–183. ACM New York, NY, USA, 2005.

[13] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing Web application code by static analysis and runtime protection. In *Proceedings of the 13th conference on World Wide Web*, pages 40–52. ACM Press, 2004.

[14] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Verifying Web applications using bounded model checking. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, pages 199–208. IEEE Computer Society, 2004.

[15] IMMUNITY. Spike. http://www.immunitysec.com/resources-freesoftware.shtml, 2004.

[16] R. Johnson and D. Wagner. Finding user/kernel pointer bugs with type inference. In *Proceedings of the 2004 Usenix Security Conference*, pages 119–134, 2004.

[17] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting Web application vulnerabilities. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 258–263. IEEE Computer Society, 2006.

[18] N. Jovanovic, C. Kruegel, and E. Kirda. Precise alias analysis for static detection of Web application vulnerabilities. In *Proceedings of the 2006 Workshop on Program-*

ming Languages and Analysis for Security, pages 27–36. ACM New York, NY, USA, 2006.

[19] S. Kals, E. Kirda, C. Kruegel, and N. Jovanovic. SecuBat: A Web vulnerability scanner. In *Proceedings of the 15th International Conference on World Wide Web*, pages 247–256. ACM New York, NY, USA, 2006.

[20] M.S. Lam, M. Martin, B. Livshits, and J. Whaley. Securing Web applications with static and dynamic information flow tracking. In *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 3–12. ACM New York, NY, USA, 2005.

[21] V.B. Livshits and M. S. Lam. Finding security vulnerabilities in Java applications with static analysis. In *Proceedings of the 14th Conference on USENIX Security Symposium*. USENIX Association Berkeley, CA, USA, 2005.

[22] J. Martin and B. Christian. SMask: Preventing injection attacks in Web applications by approximating automatic data/code separation. In *22nd ACM Symposium on Applied Computing*, pages 284–291, 2007.

[23] M. Martin, B. Livshits, and M. 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*, pages 365–383. ACM New York, NY, USA, 2005.

[24] F. Mavituna. SQL injection cheat sheet, version 1.4. http://ferruh.mavituna.com/makale/sql-injection-cheatsheet/, 2007.

[25] Y. Minamide. Static approximation of dynamically generated Web pages. In *Proceedings of the 14th International Conference on World Wide Web*, pages 432–441. ACM New York, NY, USA, 2005.

[26] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of 38th Design Automation Conference*, pages 530–535. IEEE, 2001.

[27] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 228–241, 1999.

[28] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium*, pages 131–144. San Diego, IEEE Computer Society, 2005.

[29] A. Nguyen-tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening Web applications using precise tainting. In *20th IFIP International Information Security Conference*, pages 295–307, 2005.

[30] OWASP. Top 10 2007. http://www.owasp.org/index.php/Top_10_2007, 2008.

[31] PHP. PHP: Hypertext preprocessor. http://www.php.net, 2008.

[32] T. Pietraszek and C.V. Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Recent Advances in Intrusion Detection*, pages 124–145, 2005.

[33] RSnake. XSS (cross site scripting) cheat sheet. http://ha.ckers.org/xss.html/, 2007.

[34] SAMATE. NIST. http://samate.nist.gov/index.php/Source_Code_Security_Analyzers.html, 2009.

[35] SecurityFocus. Flawed USC admissions site allowed access to applicant data. http://www.securityfocus.com/news/11239, 2005.

[36] U. Shankar, K. Talwar, J.S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th USENIX Security Symposium*, pages 201–220. USENIX Association Berkeley, CA, USA, 2001.

[37] TIOBE Software. TIOBE programming community index for June 2009. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html, 2009.

[38] R.E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. In *IEEE Transactions on Software Engineering*, pages 157–171. IEEE Press, 1986.

[39] Syhunt. Product updates. http://www.syhunt.com/, 2009.

[40] Symantec. Internet security threat report volume XIII: April, 2008. http://www.symantec.com/business/theme.jsp?themeid=threatreport, 2008.

[41] VNUNET. Monster.com hid site hack for five days. http://www.vnunet.com/vnunet/news/2197408/monster-keptreach-secret-five, 2007.

[42] G. Wassermann and Z. Su. Sound and precise analysis of Web applications for injection vulnerabilities. In *Proceedings of the 2007 PLDI Conference*, pages 22–41. ACM New York, NY, USA, 2007.

[43] J. Whaley and M.S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 131–144. ACM New York, NY, USA, 2004.

[44] J. Wilander and M. Kamkar. A comparison of publicly available tools for static intrusion prevention. In *Proceedings of 7th Nordic Workshop on Secure IT Systems*, pages 68–84, 2002.

[45] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the 15th Conference on USENIX Security Symposium*, pages 179–192. USENIX Association Berkeley, CA, USA, 2006.

Appendix

We design 90 benchmark cases to implement static analysis in this section, and we write a description at the bottom of each of them to illustrate what vulnerabilities occur and how vulnerabilities propagate in the code.

#1: XSS.php
01: <?php 02: $name = $_GET['name']; **03: echo "Welcome $name";** 04: ?>
The result will be printed, by function **echo**, at line 3 through the parameter, named **$name** at line 2, got from request method **GET**, and XSS occur at line 3 if Web site gets a malicious input.

#2: XSS_1.php
01: <?php 02: $first = $_GET['first']; 03: $a = array ('a' => $first, 'b' => 'banana', 'c' => 'cat'); **04: print_r ($a);** 05: ?>
The result will be printed, by function **print_r()**, at line 4 through the parameter, named **$name** at line 2, got from request method **GET**, and XSS occur at line 4 if Web site gets a malicious input.

#3: XSS_2.php

```
01: <?php
02:   $db = $_GET['dbName'];
03:
04:    $con = mysql_connect("localhost", "root", "svvrl");
05:   mysql_select_db($db) or die("無此資料庫: $db");
06:   $result = mysql_query("select * from stuInfo where stu_id=100001");
07:
08:   while ( $row = mysql_fetch_array($result) ) {
09:      echo $row['stu_name']."<BR>";
10:   }
11: ?>
```

Firstly, the output value wrapped in **while loop** at line 9 is from the result of database manipulation at line 6. In other words, the value printed is from the database, but XSS occurs at line 9 if the value from database is taint. Secondly, the second variable **$db** at line 5 is displayed through a user input at line 2, but XSS occurs at line 5 if Web site gets a malicious input. Thirdly, it is dangerous to hard code user name and password at line 4, if a hacker can invade server and steal out this program file.

#4: XSS_3_front.php

```
01: <html>
02: <head>
03: <meta http-equiv="Content-Type" content="text/html; charset=big5">
04: <title></title>
05: </head>
06: <body>
07:   <form name="form1" method="post" action="XSS_3_back.php">
08:     <p>The hidden value's been hidden, and prees SUBMIT if u wanna see it :
09:                 <input type='hidden' name='Hvalue' value='<?php echo
$_GET['hiddenvalue']; ?>'>
10:     </p>
11:     <p>
12:     <input type="submit" name="Submit" value="SUBMIT">
13:     </p>
14:   </form>
15: </body>
16: </html>
```

#5: XSS_3_back.php

```
01: <?php
02:    echo "The hidden value is: ".$_POST['Hvalue'];
03: ?>
```

The output value at line 2 in page **XSS_3_back.php** is displayed, by function **echo**, through the previous page **XSS_3_front.php**, but XSS occurs at line 2 in page **XSS_3_back.php** if Web site gets a malicious value from previous page.

#6: XSS_4_front.php

```
01: <html>
02: <head>
03: <meta http-equiv="Content-Type" content="text/html; charset=big5">
04: <title></title>
05: </head>
06: <body>
07:   <form name="form1" method="post" action="XSS_3_back.php">
08:    <p>The hidden value's been hidden, and prees SUBMIT if u wanna see it :
09:                <input type='hidden' name='Hvalue' value='<?php echo
$_GET['hiddenvalue']; ?>'>
10:    </p>
11:    <p>
12:    <input type="submit" name="Submit" value="SUBMIT">
13:    </p>
14:   </form>
15: </body>
16: </html>
```

#7: XSS_4_back.php

```
01: <?php
02:    print_r ("The hidden value is: ".$_POST['Hvalue']);
03: ?>
```

The output value at line 2 in page **XSS_4_back.php** is displayed, by function **print_r**, through the previous page **XSS_4_front.php**, but XSS occurs at line 2 in page **XSS_4_back.php** if Web site gets a malicious value from previous page.

#8: XSS_5.php

```
01: <?php
02:   $db = $_GET['dbName'];
03:
04:    $con = mysql_connect("localhost", "root", "svvrl");
05:   mysql_select_db($db) or die("無此資料庫: $db");
06:   $result = mysql_query("select * from stuInfo where stu_id=100002");
07:
08:   if( $row = mysql_fetch_array($result) ) {
09:      echo $row['stu_name']."<BR>";
10:   }
11: ?>
```

Firstly, the output value wrapped in **if statement** at line 9 is from the result of database manipulation at line 6. In other words, the value printed is from the database, but XSS occurs at line 9 if the value from database is taint. Secondly, the second variable **$db** at line 5 is displayed through a user input at line 2, but XSS occurs at line 5 if Web site gets a malicious input. Thirdly, it is dangerous to hard code user name and password at line 4, if a hacker can invade server and steal out this program file.

#9: XSS_6.php

```
01: <?php
02:   $db = $_GET['dbName'];
03:
04:    $con = mysql_connect("localhost", "root", "svvrl");
05:   mysql_select_db($db) or die("無此資料庫: $db");
06:   $result = mysql_query("select * from stuInfo where stu_id=100001");
07:
08:   do{
09:      echo $row['stu_name']."<BR>";
10:   }while ( $row = mysql_fetch_array($result));
11: ?>
```

Firstly, the output value wrapped in **do-while loop** at line 9 is from the result of database manipulation at line 6. In other words, the value printed is from the database, but XSS occurs at line 9 if the value from database is taint. Secondly, the second variable **$db** at line 5 is displayed through a user input at line 2, but XSS occurs at line 5 if Web site gets a malicious input. Thirdly, it is dangerous to hard code user name and password at line 4, if a hacker can invade server and steal out this program file.

#10: XSS_7.php

```
01: <?php
02:   //$db = $_GET['dbName'];
03:   $db = 'tabitha';
04:
05:    $con = mysql_connect("localhost", "root", "svvrl");
06:   mysql_select_db($db) or die("無此資料庫: $db");
07:   $result = mysql_query("select * from stuInfo where stu_id=100002");
08:
09:   while( $row = mysql_fetch_array($result) ) {
10:      echo $row['stu_name']."<BR>";
11:   }
12: ?>
```

Firstly, the output value wrapped in **while loop** at line 10 is from the result of database manipulation at line 7. In other words, the value printed is from the database, but XSS occurs at line 10 if the value from database is taint. Secondly, it is dangerous to hard code user name and password at line 5, if a hacker can invade server and steal out this program file.

#11: XSS_8.php

```
01: <?php
02:   //$db = $_GET['dbName'];
03:   $db = 'tabitha';
04:    $con = mysql_connect("localhost", "root", "svvrl");
05:   mysql_select_db($db) or die("無此資料庫: $db");
06:   $result = mysql_query("select * from stuInfo where stu_id=100002");
07:
08:   if( $row = mysql_fetch_array($result) ) {
09:      echo $row['stu_name']."<BR>";
10:   }
11: ?>
```

Firstly, the output value wrapped in **if statement** at line 9 is from the result of database manipulation at line 6. In other words, the value printed is from the database, but XSS occurs at line 9 if the value from database is taint. Secondly, it is dangerous to hard code user name and password at line 4, if a hacker can invade server and steal out this program file.

#12: XSS_9.php

```
01: <?php
02:   //$db = $_GET['dbName'];
03:   $db = 'tabitha';
04:
05:    $con = mysql_connect("localhost", "root", "svvrl");
06:   mysql_select_db($db) or die("無此資料庫: $db");
07:   $result = mysql_query("select * from stuInfo where stu_id=100006");
08:
09:   do{
10:      echo $row['stu_name']."<BR>";
11:   }while ( $row = mysql_fetch_array($result));
12: ?>
```

Firstly, the output value wrapped in **do-while statement** at line 10 is from the result of database manipulation at line 7. In other words, the value printed is from the database, but XSS occurs at line 10 if the value from database is taint. Secondly, it is dangerous to hard code user name and password at line 5, if a hacker can invade server and steal out this program file.

#13: VVXSS.php

```
01: <?php
02:   $source = $_GET['name'];
03:   $vv = 'source';
04:    echo "Welcome ".$$vv;
05: ?>
```

The result, with variable variable structure, will be printed, by function **echo**, at line 4 through the parameter, named **$srouce** at line 2, got from request method **GET**, and XSS occur at line 4 if Web site gets malicious input.

#14: VVXSS_1.php

```
01: <?php
02:   $source = $_GET['name'];
03:   $vv = 'source';
04:    print_r ("Welcome ".$$vv);
05: ?>
```

The result, with variable variable structure, will be printed, by function **print_r()**, at line 4 through the parameter, named **$srouce** at line 2, got from request method **GET**, and XSS occur at line 4 if Web site gets malicious input.

#15: VVXSS_2.php

```
01: <?php
02:   $dbName = $_GET['dbName'];
03:   $db ='dbName';
04:    $con = mysql_connect("localhost", "root", "svvrl");
05:   mysql_select_db($$db) or die("無此資料庫: ".$$db);
06:   $result = mysql_query("select * from stuInfo where stu_id=100001");
07:
08:   while ( $row = mysql_fetch_array($result) ) {
09:      echo $row['stu_name']."<BR>";
10:   }
11: ?>
```

Firstly, the output value wrapped in **while statement** at line 9 is from the result of database manipulation at line 6. In other words, the value printed is from the database, but XSS occurs at line 9 if the value from database is taint. Secondly, the second **variable variable $$db** at line 5 is displayed through user input at line 2, but XSS occurs at line 5 if Web site gets malicious input. Thirdly, user name and password are hard-coded at line 4 is dangerous, if hacker can invade server and steal out this program file.

#16: VVXSS_3_front.php

```
01: <html>
02: <head>
03: <meta http-equiv="Content-Type" content="text/html; charset=big5">
04: <title></title>
05: </head>
06: <body>
07:   <form name="form1" method="post" action="VVXSS_3_back.php">
08:     <p>The hidden value has been hidden, and prees SUBMIT if u wanna see it :
09:             <input type='hidden' name='Hvalue' value='<?php echo $_GET['hiddenvalue']; ?>'>
10:     </p>
11:     <p>
12:     <input type="submit" name="Submit" value="SUBMIT">
13:     </p>
14:   </form>
15: </body>
16: </html>
```

#17: VVXSS_3_back.php

```
01: <?php
02:   $source = $_POST['Hvalue'];
03:   $vv = 'source';
04:    echo "The hidden value is: ".$$vv;
05: ?>
```

The output value **$$vv** at line 4 in page **VVXSS_3_back.php** is displayed, by function **echo**, through the previous page **VVXSS_3_front.php**, but XSS occurs at line 4 in page **VVXSS_3_back.php** if Web site gets malicious value from previous page.

#18: VVXSS_4_front.php

```
01: <html>
02: <head>
03: <meta http-equiv="Content-Type" content="text/html; charset=big5">
04: <title></title>
05: </head>
06: <body>
07:   <form name="form1" method="post" action="VVXSS_4_back.php">
08:     <p>The hidden value has been hidden, and prees SUBMIT if u wanna see it :
09:             <input type='hidden' name='Hvalue' value='<?php echo $_GET['hiddenvalue']; ?>'>
10:     </p>
11:     <p>
12:     <input type="submit" name="Submit" value="SUBMIT">
13:     </p>
14:   </form>
15: </body>
16: </html>
```

#19: VVXSS_4_back.php

```
01: <?php
02:   $source = $_POST['Hvalue'];
03:   $vv = 'source';
04:   print_r ("The hidden value is: ".$$vv);
05: ?>
```

The output value **$$vv** at line 4 in page **VVXSS_4_back.php** is displayed, by function **print_r()**, through the previous page **VVXSS_4_front.php**, but XSS occurs at line 4 in page **VVXSS_4_back.php** if Web site gets a malicious value from previous page.

#20: VVXSS_5.php

```
01: <?php
02:   $dbName = $_GET['dbName'];
03:   $db ='dbName';
04:    $con = mysql_connect("localhost", "root", "svvrl");
05:   mysql_select_db($$db) or die("無此資料庫: ".$$db);
06:   $result = mysql_query("select * from stuInfo where stu_id=100001");
07:
08:   if( $row = mysql_fetch_array($result) ) {
09:      echo $row['stu_name']."<BR>";
10:   }
11: ?>
```

Firstly, the output value wrapped in **if statement** at line 9 is from the result of database manipulation at line 6. In other words, the value printed is from the database, but XSS occurs at line 9 if the value from database is taint. Secondly, the second **variable variable $$db** at line 5 is displayed through a user input at line 2, but XSS occurs at line 5 if Web site gets a malicious input. Thirdly, it is dangerous to hard code user name and password at line 4, if a hacker can invade server and steal out this program file.

#21: VVXSS_6.php

```
01: <?php
02:   $dbName = $_GET['dbName'];
03:   $db ='dbName';
04:    $con = mysql_connect("localhost", "root", "svvrl");
05:   mysql_select_db($$db) or die("無此資料庫: ".$$db);
06:   $result = mysql_query("select * from stuInfo where stu_id=100001");
07:
08:   do{
09:      echo $row['stu_name']."<BR>";
10:   }while ( $row = mysql_fetch_array($result));
11: ?>
```

Firstly, the output value wrapped in **do-while loop** at line 9 is from the result of database manipulation at line 6. In other words, the value printed is from the database, but XSS occurs at line 9 if the value from database is taint. Secondly, the second **variable variable $$db** at line 5 is displayed through a user input at line 2, but XSS occurs at line 5 if Web site gets a malicious input. Thirdly, it is dangerous to hard code user name and password at line 4, if a hacker can invade server and steal out this program file.

#22: VVXSS_7.php

```
01: <?php
02:   $untaint1 = "apple";
03:   $tmp = $untaint1;
04:   $str = "untaint1";
05:   $$str = $_GET["attack"];
06:    echo $untaint1;
07: ?>
```

The output value at line 6 effected by variable variable $$str at line 5, although $untaint1 is not the same as $$str intuitively. XSS occurs if the Web site gets a malicious input.

#23: VVXSS_8.php

```
01: <?php
02:   $taint = $_GET["attack"];
03:   $untaint = "constant string";
04:   $choice = $_GET["choice"];
05:
06:   if ($choice==1)
07:     $str = "taint";
08:   else
09:     $str = "untaint";
10:
11:    echo $$str;
12: ?>
```

If the value of $choice is 1, the values of $$str and $taint is the same. XSS occurs if the Web site takes a malicious input.

#24: VVXSS_9.php

```
01: <?php
02:   //$dbName = $_GET['dbName'];
03:   $dbName = 'tabitha';
04:   $db ='dbName';
05:    $con = mysql_connect("localhost", "root", "svvrl");
06:   mysql_select_db($$db) or die("無此資料庫: ".$$db);
07:   $result = mysql_query("select * from stuInfo where stu_id=100001");
08:
09:   while ( $row = mysql_fetch_array($result) ) {
10:      echo $row['stu_name']."<BR>";
11:   }
12: ?>
```

Firstly, the output value wrapped in **while loop** at line 10 is from the result of database manipulation at line 7. In other words, the value printed is from the database, but XSS occurs at line 10 if the value from database is taint. Secondly, it is dangerous to hard code user name and password at line 5, if a hacker can invade server and steal out this program file.

#25: VVXSS_10.php

```
01: <?php
02:   //$dbName = $_GET['dbName'];
03:   $dbName = 'tabitha';
04:   $db ='dbName';
05:    $con = mysql_connect("localhost", "root", "svvrl");
06:   mysql_select_db($$db) or die("無此資料庫: ".$$db);
07:   $result = mysql_query("select * from stuInfo where stu_id=100001");
08:
09:   if( $row = mysql_fetch_array($result) ) {
10:      echo $row['stu_name']."<BR>";
11:   }
12: ?>
```

Firstly, the output value wrapped in **if statement** at line 10 is from the result of database manipulation at line 7. In other words, the value printed is from the database, but XSS occurs at line 10 if the value from database is taint. Secondly, it is dangerous to hard code user name and password at line 5, if a hacker can invade server and steal out this program file.

#26: VVXSS_11.php

```
01: <?php
02:   //$dbName = $_GET['dbName'];
03:   $dbName = 'tabitha';
04:   $db ='dbName';
05:    $con = mysql_connect("localhost", "root", "svvrl");
06:   mysql_select_db($$db) or die("無此資料庫: ".$$db);
07:   $result = mysql_query("select * from stuInfo where stu_id=100001");
08:
09:   do{
10:      echo $row['stu_name']."<BR>";
11:   }while ( $row = mysql_fetch_array($result));
12: ?>
```

Firstly, the output value wrapped in **do-while loop** at line 10 is from the result of database manipulation at line 7. In other words, the value printed is from the database, but XSS occurs at line 10 if the value from database is taint. Secondly, it is dangerous to hard code user name and password at line 5, if a hacker can invade server and steal out this program file.

#27: SIXSS.php

```
01: <?php
02:   $name = $_GET['name'];
03:
04:   $myArray=array ("one"=>$name);
05:   $arrindex = array("one");
06:
07:   foreach($arrindex as $val)
08:      echo "Welcome ".$myArray[$val];
09: ?>
```

The result, with string index structure, will be printed, by function **echo**, at line 8 through the parameter, named **$name** at line 2, got from request method **GET**, and XSS occur at line 8 if Web site gets a malicious input.

#28: SIXSS_1.php

```
01: <?php
02:   $name = $_GET['name'];
03:
04:   $myArray=array ("one"=>$name);
05:   $arrindex = array("one");
06:
07:   foreach($arrindex as $val)
08:      print_r ("Welcome ".$a[$val]);
09: ?>
```

The result, with string index structure, will be printed, by function **print_r**, at line 8 through the parameter, named **$name** at line 2, got from request method **GET**, and XSS occur at line 8 if Web site gets a malicious input.

#29: SIXSS_2.php

```
01: <?php
02:   $db = $_GET['dbName'];
03:   $a = array ('one' => $db, 'two' => 'banana', 'three' => 'cat');
04:   $arrindex = array('one');
05:
06:   foreach($arrindex as $val)
07:   {
08:      $con = mysql_connect("localhost", "root", "svvrl");
09:     mysql_select_db($a[$val]) or die("無此資料庫: $a[$val]");
10:     $result = mysql_query("select * from stuInfo where stu_id=100001");
11:
12:     while ( $row = mysql_fetch_array($result) ) {
13:        echo $row['stu_name']."<BR>";
14:     }
15:   }
16: ?>
```

Firstly, the output value wrapped in **while loop** at line 13 is from the result of database manipulation at line 10. In other words, the value printed is from the database, but XSS occurs at line 13 if the value from database is taint. Secondly, the second variable **$a[$val]** at line 9 is displayed through a user input at line 2, but XSS occurs at line 9 if Web site gets a malicious input. Thirdly, it is dangerous to hard code user name and password at line 8, if a hacker can invade server and steal out this program file.

#30: SIXSS_3_front.php

```
01: <html>
02: <head>
03: <meta http-equiv="Content-Type" content="text/html; charset=big5">
04: <title></title>
05: </head>
06: <body>
07:   <form name="form1" method="post" action="SIXSS_3_back.php">
08:     <p>The hidden value has been hidden, and prees SUBMIT if u wanna see it：
09:                 <input type='hidden' name='Hvalue' value='<?php echo $_GET['hiddenvalue']; ?>'>
10:     </p>
11:     <p>
12:     <input type="submit" name="Submit" value="SUBMIT">
13:     </p>
14:   </form>
15: </body>
16: </html>
```

#31: SIXSS_3_back.php

```
01: <?php
02:   $name = $_POST['Hvalue'];
03:
04:   $myArray=array ("one"=>$name);
05:   $arrindex = array("one");
06:
07:   foreach($arrindex as $val)
08:     echo "The hidden value is: ".$myArray[$val];
09: ?>
```

The output value at line 8 in page **SIXSS_3_back.php** is displayed, by function **echo**, through the previous page **SIXSS_3_front.php**, but XSS occurs at line 8 in page **SIXSS_3_back.php** if Web site gets a malicious value from previous page.

#32: SIXSS_4_front.php

```
01: <html>
02: <head>
03: <meta http-equiv="Content-Type" content="text/html; charset=big5">
04: <title></title>
05: </head>
06: <body>
07:   <form name="form1" method="post" action="SIXSS_4_back.php">
08:     <p>The hidden value has been hidden, and prees SUBMIT if u wanna see it :
09:                 <input type='hidden' name='Hvalue' value='<?php echo $_GET['hiddenvalue']; ?>'>
10:     </p>
11:     <p>
12:     <input type="submit" name="Submit" value="SUBMIT">
13:     </p>
14:   </form>
15: </body>
16: </html>
```

#33: SIXSS_4_back.php

```
01: <?php
02:   $name = $_POST['Hvalue'];
03:
04:   $myArray=array ("one"=>$name);
05:   $arrindex = array("one");
06:
07:   foreach($arrindex as $val)
08:     print_r ("The hidden value is: ".$myArray[$val]);
09: ?>
```

The output value at line 8 in page **SIXSS_4_back.php** is displayed, by function **print_r()**, through the previous page **SIXSS_4_front.php**, but XSS occurs at line 8 in page **SIXSS_4_back.php** if Web site gets a malicious value from previous page.

#34: SIXSS_5.php

```
01: <?php
02:   $db = $_GET['dbName'];
03:   $a = array ('one' => $db, 'two' => 'banana', 'three' => 'cat');
04:   $arrindex = array('one');
05:
06:   foreach($arrindex as $val)
07:   {
08:     $con = mysql_connect("localhost", "root", "svvrl");
09:     mysql_select_db($a[$val]) or die("無法此資料庫: $a[$val]");
10:     $result = mysql_query("select * from stuInfo where stu_id=100001");
11:
12:     if( $row = mysql_fetch_array($result) ) {
13:       echo $row['stu_name']."<BR>";
14:     }
15:   }
16: ?>
```

Firstly, the output value wrapped in **if statement** at line 13 is from the result of database manipulation at line 10. In other words, the value printed is from the database, but XSS occurs at line 13 if the value from database is taint. Secondly, the second variable **$a[$val]** at line 9 is displayed through a user input at line 2, but XSS occurs at line 9 if Web site gets a malicious input. Thirdly, it is dangerous to hard code user name and password at line 8, if a hacker can invade server and steal out this program file.

#35: SIXSS_6.php

```
01: <?php
02:   $db = $_GET['dbName'];
03:   $a = array ('one' => $db, 'two' => 'banana', 'three' => 'cat');
04:   $arrindex = array('one');
05:
06:   foreach($arrindex as $val)
07:   {
08:     $con = mysql_connect("localhost", "root", "svvrl");
09:     mysql_select_db($a[$val]) or die("無此資料庫: $a[$val]");
10:     $result = mysql_query("select * from stuInfo where stu_id=100001");
11:
12:     do{
13:       echo $row['stu_name']."<BR>";
14:     }while ( $row = mysql_fetch_array($result));
15:   }
16: ?>
```

Firstly, the output value wrapped in **do-while loop** at line 13 is from the result of database manipulation at line 10. In other words, the value printed is from the database, but XSS occurs at line 13 if the value from database is taint. Secondly, the second variable **$a[$val]** at line 9 is displayed through a user input at line 2, but XSS occurs at line 9 if Web site gets a malicious input. Thirdly, it is dangerous to hard code user name and password at line 8, if a hacker can invade server and steal out this program file.

#36: SIXSS_7.php

```
01: <?php
02:   //$db = $_GET['dbName'];
03:   $db = 'tabitha';
04:
05:   $a = array ('one' => $db, 'two' => 'banana', 'three' => 'cat');
06:   $arrindex = array('one');
07:
08:   foreach($arrindex as $val)
09:   {
10:     $con = mysql_connect("localhost", "root", "svvrl");
11:     mysql_select_db($db) or die("無此資料庫: $db");
12:     $result = mysql_query("select * from stuInfo where stu_id=100002");
13:
14:     while( $row = mysql_fetch_array($result) ) {
15:       echo $row['stu_name']."<BR>";
16:     }
17:   }
18: ?>
```

Firstly, the output value wrapped in **while loop** at line 15 is from the result of database manipulation at line 12. In other words, the value printed is from the database, but XSS occurs at line 15 if the value from database is taint. Secondly, it is dangerous to hard code user name and password at line 10, if a hacker can invade server and steal out this program file.

#37: SIXSS_8.php

```
01: <?php
02:   //$db = $_GET['dbName'];
03:   $db = 'tabitha';
04:
05:   $a = array ('one' => $db, 'two' => 'banana', 'three' => 'cat');
06:   $arrindex = array('one');
07:
08:   foreach($arrindex as $val)
09:   {
10:     $con = mysql_connect("localhost", "root", "svvrl");
11:     mysql_select_db($db) or die("無此資料庫: $db");
12:     $result = mysql_query("select * from stuInfo where stu_id=100002");
13:
14:     if( $row = mysql_fetch_array($result) ) {
15:       echo $row['stu_name']."<BR>";
16:     }
17:   }
18: ?>
```

Firstly, the output value wrapped in **if statement** at line 15 is from the result of database manipulation at line 12. In other words, the value printed is from the database, but XSS occurs at line 15 if the value from database is taint. Secondly, it is dangerous to hard code user name and password at line 10, if a hacker can invade server and steal out this program file.

#38: SIXSS_9.php

```
01: <?php
02:   //$db = $_GET['dbName'];
03:   $db = 'tabitha';
04:
05:   $a = array ('one' => $db, 'two' => 'banana', 'three' => 'cat');
06:   $arrindex = array('one');
07:
08:   foreach($arrindex as $val)
09:   {
10:     $con = mysql_connect("localhost", "root", "svvrl");
11:     mysql_select_db($db) or die("無此資料庫: $db");
12:     $result = mysql_query("select * from stuInfo where stu_id=100002");
13:
14:     do{
15:       echo $row['stu_name']."<BR>";
16:     }while ( $row = mysql_fetch_array($result));
17:   }
18: ?>
```

Firstly, the output value wrapped in **do-while loop** at line 15 is from the result of database manipulation at line 12. In other words, the value printed is from the database, but XSS occurs at line 15 if the value from database is taint. Secondly, it is dangerous to hard code user name and password at line 10, if a hacker can invade server and steal out this program file.

#39: sqlInjection_front.php

```
01: <html><head><title>SQL Injection Test Page</title>
02: </head><body>
03:   <form name="form1" method="post" action="sqlInjection_back.php">
04:     <p>Please input the department name you want：
05:     <input name="departmentName" type="text" id="departmentName">
06:     </p><p>
07:     <input type="submit" name="Submit" value="SUBMIT">
08:     <input type="reset" name="Submit2" value="RESET">
09:     </p></form></body></html>
```

#40: sqlInjection_back.php

```
01: <?php
02:   $department = $_POST['departmentName'];
03:
04:   //一、連結資料庫
05:   //建立連線：
06:    $link = mysql_pconnect("localhost", "root", "svvrl");
07:   //選擇資料庫：
08:   mysql_select_db("tabitha") or die("無法選擇資料庫");
09:   //二、執行SQL語法
10:  // 建立SQL語法 11:   $query = "SELECT * FROM stuInfo where department
= '" .$department. "'";
12:   //送出SQL語法到資料庫系統
13:    $result = mysql_query($query) or die("無法送出" . mysql_error());
14:   // 三、取得執行SQL語法後的結果(指查詢部份)
15:   while ( $row = mysql_fetch_array($result) ) {
16:      echo $row['stu_name']."<BR>";
17:   }
18:   //四、釋放與Mysql的連線
19:   mysql_free_result($result);
20: ?>
```

Firstly, it is dangerous to hard code user name and password at line 6, if a hacker can invade server and steal out this program file. Secondly, the SQL statement at line 10 will be executed at line 13, but SQL injection occurs when Web site gets a malicious input as a part in an SQL statement. Thirdly, the output value at line 16 is from the result of database manipulation at line 13. In other words, the value printed is from the database, but XSS occurs at line 16 if the value from database is taint. Fourthly, information leakage occurs if function **mysql_error()** divulges important information at line 13.

#41: sqlInjection_1.php

```
01: <?php
02:    $link = mysql_pconnect("localhost", "root", "svvrl");
03:    mysql_select_db("tabitha") or die("無法選擇資料庫");
04:
05:    $query="SELECT stu_id, department, stu_name, stu_score FROM stuinfo WHERE stu_id='100001'";
06:    $query_result=mysql_query($query);
07:    $result=mysql_fetch_row($query_result);
08:    $source1=$result[2];
09:
10:     if(!empty($source1)){
11:
12:                  $query="SELECT stu_score FROM stuinfo WHERE stu_name='".$source1."'";
13:     $query_result = mysql_query($query);
14:
15:     while ( $row = mysql_fetch_array($query_result) ) {
16:                  echo "The student $source1 's score is: ".$row['stu_score']."<BR>";
17:     }
18:   }
19: ?>
```

Firstly, it is dangerous to hard code user name and password at line 2, if a hacker can invade server and steal out this program file. Secondly, the SQL statement at line 12 will be executed at line 13, but SQL injection occurs when Web site gets a malicious input as a part in an SQL statement. Thirdly, two output values, $source1 and $row['stu_score'], at line 16 are from the result of database manipulation at line 13. In other words, the value printed is from the database, but XSS occurs at line 16 if the value from database is taint.

#42: sqlInjection_2_front.php

```
01: <html><head>
02: <meta http-equiv="Content-Type" content="text/html; charset=big5">
03: <title>SQL Injection Test Page</title></head>
04: <body>
05:   <form name="form1" method="post" action="sqlInjection_2_back.php">
06:     <p>Id Number(e.g. 100001)：
07:     <input name="id" type="text" id="id"></p>
08:     <p>Department：
09:     <input name="department" type="text" id="department"></p>
10:     <p>Name：
11:     <input name="name" type="text" id="name"></p>
12:     <p>Score：
13:     <input name="score" type="text" id="score"></p>
14:     <p>
15:     <input type="submit" name="Submit" value="SUBMIT">
16:     <input type="reset" name="Submit2" value="RESET">
17:     </p>
18:   </form></body></html>
```

#43: sqlInjection_2_back.php

```
01: <?php
02:   $stu_id = $_POST['id'];
03:   $department = $_POST['department'];
04:   $stu_name = $_POST['name'];
05:   $stu_score = $_POST['score'];
06:
07:    $link = mysql_pconnect("localhost", "root", "svvrl");
08:   mysql_select_db("tabitha") or die("無法選擇資料庫");
09:
10:    $query="INSERT INTO stuinfo(stu_id, department, stu_name, stu_score)
VALUES ('".$stu_id."', '".$department."', '".$stu_name."', '".$stu_score."')";
11:    mysql_query($query);
12:   echo"<center>已新增資料<p></center>";
13: ?>
```

Firstly, it is dangerous to hard code user name and password at line 2, if a hacker can invade server and steal out this program file. Secondly, the SQL statement at line 10 in page **sqlInjection_2_front.php** is executed at line 11, but SQL injection occurs at line 11 in page **sqlInjection_2_back.php** if Web site gets malicious value as a part in an SQL statement from previous page.

#44: sqlInjection_3_front.php

```
01: <html>
02: <head>
03: <meta http-equiv="Content-Type" content="text/html; charset=big5">
04: <title>SQL Injection Test Page</title>
05: </head>
06: <body>
07:   <form name="form1" method="post" action="sqlInjection_3_back.php">
08:     <p>Id Number(e.g. 100001)：
09:     <input name="id" type="text" id="id">
10:     <p>Update the Score：
11:     <input name="score" type="text" id="score">
12:     </p>
13:     <p>
14:     <input type="submit" name="Submit" value="SUBMIT">
15:     <input type="reset" name="Submit2" value="RESET">
16:     </p>
17:   </form>
18: </body>
19: </html>
```

#45: sqlInjection_3_back.php

```
01: <?php
02:   $stu_id = $_POST['id'];
03:   $stu_score = $_POST['score'];
04:
05:    $link = mysql_pconnect("localhost", "root", "svvrl");
06:   mysql_select_db("tabitha") or die("無法選擇資料庫");
07:
08:    $query="UPDATE stuinfo SET stu_score = '".$stu_score."' WHERE stu_id
="".$stu_id."";
09:    mysql_query($query);
10:   echo"<center>已修改資料<p></center>";
11: ?>
```

Firstly, it is dangerous to hard code user name and password at line 5, if a hacker can invade server and steal out this program file. Secondly, the SQL statement at line 8 in page **sqlInjection_3_front.php** is executed at line 9, but SQL injection occurs at line 9 in page **sqlInjection_3_back.php** if Web site gets a malicious value as a part in an SQL statement from previous page.

#46: sqlInjection_4_front.php

```
01: <html>
02: <head>
03: <meta http-equiv="Content-Type" content="text/html; charset=big5">
04: <title>SQL Injection Test Page</title>
05: </head>
06: <body>
07:   <form name="form1" method="post" action="sqlInjection_4_back.php">
08:     <p¿Id Number(e.g. 100001)：
09:     <input name="id" type="text" id="id">
10:     <p>
11:     <input type="submit" name="Submit" value="SUBMIT">
12:     <input type="reset" name="Submit2" value="RESET">
13:     </p>
14:   </form>
15: </body>
16: </html>
```

#47: sqlInjection_4_back.php

```
01: <?php
02:   $stu_id = $_POST['id'];
03:
04:    $link = mysql_pconnect("localhost", "root", "svvrl");
05:   mysql_select_db("tabitha") or die("無法選擇資料庫");
06:
07:   $query="DROP table $dbName"
08:    mysql_query($query);
09:   echo"<center>已刪除資料表<p></center>";
10: ?>
```

Firstly, it is dangerous to hard code user name and password at line 4, if a hacker can invade server and steal out this program file. Secondly, the SQL statement at line 7 in page **sqlInjection_4_front.php** is executed at line 8, but SQL injection occurs at line 8 in page **sqlInjection_4_back.php** if Web site gets a malicious value as a part in an SQL statement from previous page.

#48: resourceInjection_front.php

```
01: <html>
02: <head>
03: <meta http-equiv="Content-Type" content="text/html; charset=big5">
04: <title>SQL Injection Test Page</title>
05: </head>
06: <body>
07:  <form name="form1" method="post" action="resourceInjection_back.php">
08:   <p>Please input the file name you want to see(1 or 2) :
09:   <input name="fileName" type="text" id="fileName">
10:   </p>
11:     <p>
12:     <input type="submit" name="Submit" value="SUBMIT">
13:     <input type="reset" name="Submit2" value="RESET">
14:     </p>
15:   </form>
16: </body>
17: </html>
```

#49: resourceInjection_back.php

```
01: <?php
02:   $fileName = $_POST['fileName'];
03:    echo "<a href=$fileName.txt>Click here to see the file</a>";
04: ?>
```

Firstly, the hyperlink statement at line 3 in page **resourceInjection_front.php** is executed through the previous page **resourceInjection_back.php**, but resource injection occurs at line 2 in page **resourceInjection_back.php** if Web site gets a malicious value as a part in hyperlink manipulation from previous page. Secondly, the output value at line 3 in page **resourceInjection_back.php** is displayed, by function **echo**, but XSS occurs at line 3 in page **resourceInjection_back.php** if Web site gets a malicious value from previous page.

#50: resourceInjection_1.php

```
01: <?php
02:   $host = $_GET['host'];
03:
04:    $con = mysql_connect($host, "root", "svvrl");
05:   mysql_select_db("tabitha") or die("無法選擇資料庫");
06:   $result = mysql_query("select * from stuInfo where stu_id=100001");
07:
08:   while ( $row = mysql_fetch_array($result) ) {
09:      echo $row['stu_name']."<BR>";
10:   }
11: ?>
```

Firstly, it is dangerous to hard code user name and password at line 4, if a hacker can invade server and steal out this program file. Secondly, the value of a host address at line 4 is from a user input at line 2, but resource injection occurs if Web site gets a malicious value. Thirdly, the output value at line 9 is from the result of database manipulation at line 6. In other words, the value printed is from the database, but XSS occurs at line 9 if the value from database is taint.

#51: resourceInjection_2.php

```
01: <?php
02:   $fileName = $_GET['fileN'];
03:   rmdir($fileName);
04:   echo "you've removed the folder $fileName!";
05: ?>
```

Firstly, the value of file manipulation at line 4 is from a user input at line 2, but resource injection occurs if Web site gets a malicious value. Secondly, the variable **$fileName** at line 4 is displayed through a user input at line 2, but XSS occurs at line 4 if Web site gets a malicious input.

#52: resourceInjection_3.php

```
01: <?php
02:   $fileName = $_GET[filename];
03:   $file = file_get_contents($fileName, FILE_USE_INCLUDE_PATH);
04:   echo $file;
05: ?>
```

Firstly, the value of file manipulation at line 3 is from user input at line 2, but resource injection occurs if Web site gets a malicious value. Secondly, the variable **$fileName** at line 4 is displayed through contents of a file at line 3, but XSS occurs at line 4 if contents of a file are tainted.

#53: resourceInjection_4.php

```
01: <?php
02:   $FileName = $_GET['filename'];
03:   if (touch($FileName)) {
04:     echo "$FileName modification time has been changed to present time";
05:   } else {
06:     echo "Sorry, could not change modification time of $FileName";
07:   }
08: ?>
```

Firstly, the value of file manipulation at line 3 is from user input at line 2, but resource injection occurs if Web site gets a malicious value. Secondly, two variables, **$fileName** at line 4 and line 6, are displayed through a user input at line 2, but XSS occurs at line 4 or line 6 if Web site gets a malicious input.

#54: allow_url_fopen_Enabled.php

```
01: <?php
02:   $file = fopen ($_GET["file"], "r");
03:   if (!$file) {
04:     echo "No this file";
05:   }
06:   while (!feof ($file)) {
07:     $line = fgets ($file, 1024);
08:     echo $line."<br/>";
09:   }
10:   fclose($file);
11: ?>
```

Firstly, the value of file manipulation at line 2 is from a user input at the same line, but resource injection occurs if Web site gets a malicious value. Secondly, the variable **$line** at line 8 is displayed through contents of a file at line 2, but XSS occurs at line 4 if contents of a file ia tainted.

#55: CommandInjection.php

```
01: <?php
02:   $statement = $_GET['statement'];
03:   echo "The command $statment is: ".exec($statement);
04: ?>
```

Firstly, a command statement is executed at line 3, but command injection occurs if Web site gets a malicious command statement at line 2. Secondly, the variable **$statement** at line 3 is displayed through a user input at line 2, but XSS occurs at line 3 if Web site gets a malicious input.

#56: CommandInjection_1.php

```
01: <?php
02:   $statement = $_GET['statement'];
03:   echo passthru($statement);
04: ?>
```

A command statement is executed at line 3, but command injection occurs if Web site gets a malicious command statement at line 2.

#57: requirevar.php

```
01: <?php
02:   $IMdepartment = 'IM';
03: ?>
```

#58: require.php

```
01: <?php
02:   $includePage = $_GET['includePage'];
03:    require $includePage;
04:   //一、連結資料庫
05:   //建立連線：
06:    $link = mysql_pconnect("localhost", "root", "svvrl");
07:   //選擇資料庫：
08:   mysql_select_db("tabitha") or die("無法選擇資料庫");
09:   //二、執行SQL語法
10:   // 建立SQL語法
11:  $query = "SELECT * FROM stuInfo where department = '" .$IMdepartment. "'";
12:   //送出SQL語法到資料庫系統
13:   $result = mysql_query($query) or die("無法送出" . mysql_error( ));
14:   // 三、取得執行SQL語法後的結果(指查詢部份)
15:   while ( $row = mysql_fetch_array($result) ) {
16:      echo $row['stu_name']."<BR>";
17:   }
18:   //四、釋放與Mysql的連線
19:   mysql_free_result($result);
20: ?>
```

Firstly, malicious file inclusion occurs if Web site gets a malicious input as a file name though function **require** at line 3. Secondly, it is dangerous to hard code user name and password at line 6, if a hacker can invade server and steal out this program file. Thirdly, the SQL statement at line 11 will be executed at line 13, but SQL injection occurs when Web site gets a malicious input as a part in an SQL statement. Fourthly, information leakage occurs if function **mysql_error()** divulges important information at line 13. Fifthly, the output value at line 16 is from the result of database manipulation at line 13. In other words, the value printed is from the database, but XSS occurs at line 16 if the value from database is taint.

#57: requirevar.php

```
01: <?php
02:   $IMdepartment = 'IM';
03: ?>
```

#59: requireOnce.php

```
01: <?php
02:   $includePage = $_GET['includePage'];
03:    require_once $includPage;
04:   //一、連結資料庫
05:   //建立連線：
06:    $link = mysql_pconnect("localhost", "root", "svvrl");
07:   //選擇資料庫：
08:   mysql_select_db("tabitha") or die("無法選擇資料庫");
09:   //二、執行SQL語法
10:   // 建立SQL語法
11:  $query = "SELECT * FROM stuInfo where department = '" .$IMdepartment. "'";
12:   //送出SQL語法到資料庫系統
13:   $result = mysql_query($query) or die("無法送出" . mysql_error( ));
14:   // 三、取得執行SQL語法後的結果(指查詢部份)
15:   while ( $row = mysql_fetch_array($result) ) {
16:      echo $row['stu_name']."<BR>";
17:   }
18:   //四、釋放與Mysql的連線
19:   mysql_free_result($result);
20: ?>
```

Firstly, malicious file inclusion occurs if Web site gets malicious input as a file name though function **require_once** at line 3. Secondly, it is dangerous to hard code user name and password at line 6, if a hacker can invade server and steal out this program file. Thirdly, the SQL statement at line 11 will be executed at line 13, but SQL injection occurs when Web site get a malicious input as a part in an SQL statement. Fourthly, information leakage occurs if function **mysql_error()** divulges important information at line 13. Fifthly, the output value at line 16 is from the result of database manipulation at line 13. In other words, the value printed is from the database, but XSS occurs at line 16 if the value from database is taint.

#60: includevar.php

```
01: <?php
02:   $IMdepartment = 'IM';
03: ?>
```

#61: include.php

```
01: <?php
02:   $includePage = $_GET['includePage'];
03:    include $includePage;
04:   //一、連結資料庫
05:   //建立連線：
06:    $link = mysql_pconnect("localhost", "root", "svvrl");
07:   //選擇資料庫：
08:   mysql_select_db("tabitha") or die("無法選擇資料庫");
09:   //二、執行SQL語法
10:   // 建立SQL語法
11:  $query = "SELECT * FROM stuInfo where department = '" .$IMdepartment. "'";
12:   //送出SQL語法到資料庫系統
13:   $result = mysql_query($query) or die("無法送出" . mysql_error( ));
14:   // 三、取得執行SQL語法後的結果(指查詢部份)
15:   while ( $row = mysql_fetch_array($result) ) {
16:      echo $row['stu_name']."<BR>";
17:   }
18:   //四、釋放與Mysql的連線
19:   mysql_free_result($result);
20: ?>
```

Firstly, malicious file inclusion occurs if Web site gets malicious input as a file name though function **include** at line 3. Secondly, it is dangerous to hard code user name and password at line 6, if a hacker can invade server and steal out this program file. Thirdly, the SQL statement at line 11 will be executed at line 13, but SQL injection occurs when Web site gets a malicious input as a part in an SQL statement. Fourthly, information leakage occurs if function **mysql_error()** divulges important information at line 13. Fifthly, the output value at line 16 is from the result of database manipulation at line 13. In other words, the value printed is from the database, but XSS occurs at line 16 if the value from database is taint.

#60: includevar.php

```
01: <?php
02:   $IMdepartment = 'IM';
03: ?>
```

#62: include_Once.php

```
01: <?php
02:   $includePage = $_GET['includePage'];
03:    include_once $includePage;
04:   //一、連結資料庫
05:   //建立連線：
06:    $link = mysql_pconnect("localhost", "root", "svvrl");
07:   //選擇資料庫：
08:   mysql_select_db("tabitha") or die("無法選擇資料庫");
09:   //二、執行SQL語法
10:   // 建立SQL語法
11:  $query = "SELECT * FROM stuInfo where department = '" .$IMdepartment.
"'";
12:   //送出SQL語法到資料庫系統
13:   $result = mysql_query($query) or die("無法送出" . mysql_error( ));
14:   // 三、取得執行SQL語法後的結果(指查詢部份)
15:   while ( $row = mysql_fetch_array($result) ) {
16:      echo $row['stu_name']."<BR>";
17:   }
18:   //四、釋放與Mysql的連線
19:   mysql_free_result($result);
20: ?>
```

Firstly, malicious file inclusion occurs if Web site gets malicious input as a file name though function **include_once** at line 3. Secondly, it is dangerous to hard code user name and password at line 6, if a hacker can invade server and steal out this program file. Thirdly, the SQL statement at line 11 will be executed at line 13, but SQL injection occurs when Web site gets a malicious input as a part in an SQL statement. Fourthly, information leakage occurs if function **mysql_error()** divulges important information at line 13. Fifthly, the output value at line 16 is from the result of database manipulation at line 13. In other words, the value printed is from the database, but XSS occurs at line 16 if the value from database is taint.

#63: evalInjection_front.php

```
01: <html>
02: <head>
03: <meta http-equiv="Content-Type" content="text/html; charset=big5">
04: <title></title>
05: </head>
06: <body>
07:   <form name="form1" method="post" action="evalInjection_back.php">
08:     <p>show 'This is a(n)
09:       <input name="aString" type="text" id="aString">' in the next page. (Please type a word)
10:     </p>
11:     <p>
12:     <input type="submit" name="Submit" value="SUBMIT">
13:     </p>
14:   </form>
15: </body>
16: </html>
```

#64: evalInjection_back.php

```
01: <?php
02:   $theString = $_POST['aString'];
03:   $str = 'This is a(n) $theString.';
04:   echo "Before evaluating: ".$str. "<br/>";
05:    eval("\$str = \"$str\";");
06:    echo "After evaluating: ".$str. "<br/>";
07: ?>
```

A statement is associated with function **eval()** is evaluated at line 5. Eval injection occurs if Web site takes a malicious input as a part of a statement in eval().

#65: UnrestrictedFileUpload_front.php

```
01: <form enctype="multipart/form-data" action="UnrestrictedFileUpload_back.php" method="POST">
02:   Please choose a file uploaded: <input name="uploaded" type="file" /><br />
03:   <input type="submit" value="Upload" />
04: </form>
```

#66:UnrestrictedFileUpload_back.php

```
01: <?php
02:   $target = "upload/";
03:   $target = $target . basename( $_FILES['uploaded']['name']) ;
04:   if(move_uploaded_file($_FILES['uploaded']['tmp_name'], $target))
05:   {
06:     echo "The file ". basename( $_FILES['uploadedfile']['name']). " has been uploaded";
07:   }
08:   else {
09:     echo "Sorry, there was a problem uploading your file.";
10:   }
11: ?>
```

Firstly, the value of file manipulation at line 4 is from global variable at line 3, but resource injection occurs if Web site gets a malicious value. Secondly, the variable **$FILES['uploadedfile']['name']** at line 6 is displayed, but XSS occurs at line 6 if the value of global variable is taint.

#67: PathTraversal_front.php

```
01: <html>
02: <head>
03: <meta http-equiv="Content-Type" content="text/html; charset=big5">
04: <title></title>
05: </head>
06: <body>
07:   <form name="form1" method="post" action="PathTraversal_back.php">
08:     <p>Please input the file name you want to see(any extension file) :
09:     <input name="fileName" type="text" id="fileName">
10:     </p>
11:     </p>
12:     <p>
13:     <input type="submit" name="Submit" value="SUBMIT">
14:     </p>
15:   </form>
16: </body>
17: </html>
```

#68: PathTraversal_back.php

```
01: <?php
02:   $filename = $_POST['fileName'];
03:   $handle = fopen($filename,"r");
04:   $amt = fread($handle, filesize($filename));
05:   echo $amt;
06: ?>
```

Firstly, the values of file manipulation at line 3 and line 4 are from a user input at line 2, but resource injection occurs if Web site gets malicious value. Secondly, the output value at line 5 is from contents of a file. In other words, the value printed is from contents of a file, but XSS occurs at line 5 if the value from the file is taint.

#69: PathTraversal_1_front.php

```
01: <html>
02: <head>
03: <meta http-equiv="Content-Type" content="text/html; charset=big5">
04: <title></title>
05: </head>
06: <body>
07:   <form name="form1" method="post" action="PathTraversal_1_back.php">
08:     <p>Please input the file name you want to remove(any extension file) :
09:     <input name="fileName" type="text" id="fileName">
10:     </p>
11:     </p>
12:     <p>
13:     <input type="submit" name="Submit" value="SUBMIT">
14:     </p>
15:   </form>
16: </body>
17: </html>
```

#70: PathTraversal_1_back.php

```
01: <?php
02:   $rName = $_POST['fileName'];
03:    $rFile = fopen($rName,'a+');
04:   fclose($rFile);
05:    unlink($rName);
06: ?>
```

The values of file manipulation at line 3 and line 5 are from, respectively, user input and contents of a file, but resource injection occurs if Web site gets a malicious value from users or contents of a file is tainted.

#71: informationLeak.php

```
01: <?php
02:   phpinfo();
03: ?>
```

Information leakage occurs if function **phpinfo()** divulges important information at line 2.

#72: ErrorMessageInformationLeak.php

```
01: <?php
02:   function inverse($x)
03:   {
04:     if (!$x) {
05:       throw new Exception('Division by zero.');
06:     }
07:   else return 1/$x;
08:   }
09:
10:   try {
11:     echo inverse(5) . "<br/>";
12:     echo inverse(0) . "<br/>";
13:   }
14:   catch (Exception $e) {
15:     echo 'Caught exception: '.$e->getMessage();
16:   }
17: ?>
```

Firstly, information leakage occurs if function **Exception()** or **getMessage()** divulges ,respectively, important information at line 5 and line 15. Function **inverse()** are executed at line 11 and 12, but reflection injection occurs if the function is taint.

#73: InformationLeakThroughDebugInformation.php

```
01: <?php
02:   function a() {
03:     b();
04:   }
05:
06:   function b() {
07:     echo "The backtrace is ".
08:       debug_print_backtrace();
09:   }
10:
11:   a();
12: ?>
```

Information leakage occurs if function **debug_print_backtrace()** divulged important information at line 8.

#74: FilePermissionManipulation.php

```
01: <?php
02:   $rName = $_GET['publicReport'];
03:     chmod($rName,'0755');
04: ?>
```

The value of file manipulation at line 3 is from a user input at line 2, but resource injection occurs if Web site gets a malicious value.

#75: LogForging_front.php

```
01: <html>
02: <head>
03: <meta http-equiv="Content-Type" content="text/html; charset=big5">
04: <title></title>
05: </head>
06: <body>
07:   <form name="form1" method="post" action="LogForging_back.php">
08:     <p>Please input the file name you want to remove(any extension file) :
09:     <input name="aName" type="text" id="aName">
10:     </p>
11:     <p>Please input something(we'll log it) :
12:     <input name="aNumber" type="text" id="aNumber">
13:     </p>
14:     <p>
15:     <input type="submit" name="Submit" value="SUBMIT">
16:     <input type="reset" name="Submit2" value="RESET">
17:     </p>
18:   </form>
19: </body>
20: </html>
```

#76: LogForging_back.php

```
01: <?php
02:   $name =$_GET['aName'];
03:   $number =$_GET['aNumber'];
04:
05:    error_log("User:$name input:$number", 3, "error.log");
06: ?>
```

The value of file manipulation at line 5 in page **LogForging_back.php** is from previous page named **LogForging_front.php**, but resource injection occurs if Web site gets a malicious value from the previous page.

#77: LogForging_1_front.php

```
01: <html>
02: <head>
03: <meta http-equiv="Content-Type" content="text/html; charset=big5">
04: <title></title>
05: </head>
06: <body>
07:   <form name="form1" method="post" action="LogForging_1_back.php">
08:     <p>Please input your name：
09:     <input name="aName" type="text" id="aName">
10:     </p>
11:     <p>Please input something(we'll log it)：
12:     <input name="aNumber" type="text" id="aNumber">
13:     </p>
14:     <p>
15:     <input type="submit" name="Submit" value="SUBMIT">
16:     <input type="reset" name="Submit2" value="RESET">
17:     </p>
18:   </form>
19: </body>
20: </html>
```

#78: LogForging_1_back.php

```
01: <?php
02:   $name =$_POST['aName'];
03:   $number =$_POST['aNumber'];
04:   // file container where all texts are to be written
05:   $fileContainer = date("MjY").$name.'.log';
06:   // open the said file
07:    $filePointer = fopen($fileContainer,"w+");
08:   // text to be written in the file
09:   $logMsg = "Number is: ".$number
10:   // below is where the log message has been written to a file.
11:    fputs($filePointer,$logMsg);
12:   // close the open said file after writing the text
13:   fclose($filePointer);
14: ?>
```

The values of file manipulation at line 7 and 11 in page **LogForging_1_back.php** are from previous page named **LogForging_1_front.php**, but resource injection occurs if Web site gets a malicious value.

#79: VariableOverwrite.php

```
01: <?php
02:   $first="User";
03:   $str = $_SERVER['QUERY_STRING'];
04:    parse_str($str);
05:   echo $first;
06: ?>
```

Values of the variable are expected to modified through the function **parse_str()** so that the value of variable **$first** can be changed with this function.

#80: VariableOverwrite_1.php

```
01: <?php
02:   $first="User";
03:   $str = $_SERVER['QUERY_STRING'];
04:    mb_parse_str($str);
05:   echo $first;
06: ?>
```

Values of the variable are expected to modified through the function **mb_parse_str()** so that the value of variable **$first** can be changed with this function.

#81: ReflectionInjection.php

```
01: <?php
02:   function fruit($type)
03:   {
04:     echo "$type is your favorite fruit.<br/>";
05:   }
06:
07:   $fruit = $_GET['fruit'];
08:    call_user_func('fruit', $fruit);
09: ?>
```

Function **call_user_func()** is executed at line 8, but reflection injection occurs if functions invoked are taint.

#82: ReflectionInjection_1.php

```
01: <?php
02:   function fruit($type)
03:   {
04:     echo "$type is your favorite fruit.<br/>";
05:   }
06:   $func = $_GET['funcName'];
07:   $para = $_GET['paraName'];
08:    $func($para);
09: ?>
```

Function **$func** is executed at line 8, but reflection injection occurs if a function invoked is taint.

#83: NotAXSS_front.php

```
01: <html>
02:   <head>
03:   <title></title>
04:   </head>
05:   <body>
06:   <form name=form method=post action="NotAXSS_back.php">
07:     <p>The value are hidden<br/><input type=hidden value='nothing is in hidden field.' name=title>
08:     </p>
09:     <p>
10:     <input type=submit name=Submit value=NextPage>
11:     </p>
12:   </form>
13:   </body>
14:   </html>
```

#84: NotAXSS_back.php

```
01: <?php
02:   echo "The hidden value is: ".$_POST['title'];
03: ?>
```

There is no XSS, because the value at line 2 in page **NotAXSS_back.php** propagated from page **NotAXSS_front.php** is untaint (a constant string).

#85: NotAVVXSS_front.php

```
01: <html>
02:   <head>
03:   <title></title>
04:   </head>
05:   <body>
06:   <form name=form method=post action="NotAXVVSS_back.php">
07:     <p>The value are hidden<br/><input type=hidden value='nothing is in hidden field.' name=title>
08:     </p>
09:     <p>
10:     <input type=submit name=Submit value=NextPage>
11:     </p>
12:   </form>
13:   </body>
14:   </html>
```

#86: NotAVVXSS_back.php

```
01: <?php
02:   $taint = $_POST['title'];
03:   $vv = 'taint';
04:    echo "Title you insert is: ".$$vv;
05: ?>
```

There is no XSS, because the value at line 4 in page **NotAVVXSS_back.php** propagated from page **NotAVVXSS_front.php** is untaint (a constant string).

#87: NotASIXSS_front.php

```
01: <html>
02:   <head>
03:   <title></title>
04:   </head>
05:   <body>
06:   <form name=form method=post action="NotAXSISS_back.php">
07:     <p>The value are hidden<br/><input type=hidden value='nothing is in hidden field.' name=title>
08:     </p>
09:     <p>
10:     <input type=submit name=Submit value=NextPage>
11:     </p>
12:   </form>
13:   </body>
14:   </html>
```

#88: NotASIXSS_back.php

```
01: <?php
02:   $name = $_POST['title'];
03:
04:   $myArray=array ("one"=>$name);
05:   $arrindex = array("one");
06:
07:   foreach($arrindex as $val)
08:     echo "Title you insert is: ".$myArray[$val];
09: ?>
```

There is no XSS, because the value at line 8 in page **NotASIXSS_back.php** propagated from page **NotASIXSS_front.php** is untainted (a constant string).

#89: NoVulnerability_escape_string.php

```
01: <?php
02:   $link = mysql_pconnect("localhost", "root", "svvrl");
03:   mysql_select_db("tabitha") or die("無法選擇資料庫");
04:
05:   $department = $_GET['department'];
06:
07:   //escape the parameter e.g. c's -> c\\\'s
08:     $sql = sprintf("SELECT * FROM stuinfo WHERE department='%s'",
09:   mysql_real_escape_string($department));
10:
11:   $result = mysql_query($sql);
12:   while ( $row = mysql_fetch_array($result) ) {
13:     echo $row['stu_name']."<BR/>";
14:   }
15:   mysql_free_result($result);
16: ?>
```

There is no SQL injection, because the value of **$department** is escaped by function **mysql_real_escape_string()** at line 9. However, the output value at line 13 is from the result of database manipulation at line 11. In other words, the value printed is from the database, but XSS occurs at line 13 if the value from database is taint.

#90: NoVulnerability_informationLeak.php

```
01: <?php
02:   echo "There is no vulnerable point";
03: /*
04: <FORM method="POST">
05: 帳號：<input type="text" name="Account"><p>
06: 密碼：<input type="password" name="Password"><p>
07: 姓名：<input type="text" name="Name"><p>
08: 使用者類型：<input type="radio" name="Type" value="user">
09: 一般使用者：<input type="radio" name="Type" value="expert">專家<p>
10:   性 別 ：<input type="radio" name="Sex" value="male">男 性<input
type="radio" name="Sex" value="female">女性<p>
11: 年齡：<input type="text" name="Age"><p>
12: 職業：<input type="text" name="Occupation"><p>
13: <input type="submit" value="確定">
14: </FORM>
15: */
16: ?>
```

There is no information leakage, because no information divulges in comments.