Master's Thesis

Dynamic Taint Propagation Based on Dynamic String Conversion Detection

（動的な文字列変換の検出に基づく DTP）

Supervisor: Professor Shuichi Sakai

48-106425 Hiroshi Toi

GRADUATE SCHOOL OF INFORMATION SCIENCE AND TECHNOLOGY,
THE UNIVERSITY OF TOKYO

February 2012

Abstract

Currently, the security of web applications is faced with the threat of script injection attacks, such as cross-site scripting, and SQL injection. DTP (Dynamic Taint Propagation) has been established as a powerful technique for detecting script injection attacks, but current DTP systems suffer from a trade-off between false positives and false negatives. Therefore, Li et al. proposed an enhanced DTP system called SWIFT [1]. SWIFT traces memory accesses, detects string operations, and only propagates tainted information under string operations. Although the basic idea of SWIFT is quite promising, they only showed a preliminary implementation on a simulator and failed to show advantage in accuracy over Raksha [2], which is one of the most sophisticated platform DTP systems. In this paper, we implement SWIFT to PHP interpreter to put SWIFT into practical use. Moreover, we succeeded to show that SWIFT has better propagation accuracy than Raksha in real-world web applications.

Contents

1 Introduction **1**

2 Script Injection Attacks and Dynamic Taint Propagation **4**
2.1 SQL injection 4
2.2 DTPs to Script Injection Attacks 6
2.3 Command Parsing 7
2.4 Problems with Existing DTPs 7

3 SWIFT **10**
3.1 Radio Button and Text Box String Operations 10
3.2 Loop-in-Select and Select-in-Loop Structures for Table References 11
3.3 Algorithm of SWIFT 13
3.3.1 Overview 13
3.3.2 Select-in-Loop String Operation Detection 14
3.3.3 Propagation and Backtracking 16

4 Implementation of SWIFT to PHP **17**
4.1 Overview 17
4.2 Coping with Native Functions 18
4.3 PHP Interpreter internals 20
4.3.1 Relationship among Script, Opcode and Source code . . . 20
4.3.2 Variable Management 20
4.3.3 Memory Management 21
4.3.4 Declaration of zif functions 21
4.4 Acquisition of Memory Addresses 22
4.4.1 Zend Opcode Handler 22
4.4.2 Zif Functions 24

5 Evaluation **26**
5.1 Evaluation Method 26
5.1.1 Environment 26
5.1.2 Methodology 27
5.2 Taint Propagation Accuracy 27
5.2.1 String Operations 27
5.2.2 Real-World Web Applications 29
5.3 Execution Speed 31

6 Conclusion **33**

Bibliography **33**

Publications **36**

List of Figures

1.1 Increase in script injection attack 3

2.1 Example of SQL injection 5
2.2 Example of Base64 vulnerability 8

3.1 Two types of string operations 11
3.2 Loop-in-Select structure: safe 12
3.3 Select-in-Loop structure: unsafe 13
3.4 Address trace of *base64_encode* 15

4.1 General overview of our implementation 17
4.2 Relationship among script, intermediate code and source code . . 18
4.3 Zval structure 20
4.4 Data container for zval 21
4.5 zval-to-C data type conversion macros 21
4.6 Declaration of zif functions 22
4.7 Acquisition of memory addresses from zend opcode handler . . . 23
4.8 Acquisition of memory addresses from zif function 25

5.1 Qwikiwiki 1.4.1 vulnerability 29
5.2 PHP-Nuke 7.1 vulnerability 30

List of Tables

4.1 Memory-management Wrapper Functions 22

5.1 Evaluation environment 27
5.2 Results of string operation 28
5.3 Results of web applications with known vulnerabilities 31
5.4 Results of Latest web applications 31
5.5 Performance overheads 32

Chapter 1

Introduction

Along with the increase in web applications, attacks exploiting vulnerabilities of these applications have also increased. Attackers exploit various security vulnerabilities to carry out a wide variety of tasks, such as stealing secret or personal information, making a profit, or just enjoying.

In the past, the most prevalent attacks were those aimed at client applications in binary code, represented by buffer overflow attacks. The frequency of this kind of attacks, however, has abated, possibly because most of these can be prevented by NX bit and ASLR (Address Space Layout Randomization).

Instead, the most serious attacks in recent years have been ***script injection attacks*** to web servers, such as cross-site scripting (XSS), SQL injection, and directory traversal. Figure 1.1 shows the number of vulnerabilities reported to the National Vulnerability Database [3]. The figure shows vulnerabilities to script injection attacks have increased sharply in recent years.

DTP (Dynamic Taint Propagation) has been proposed to prevent these attacks. DTP systems tag data from untrusted sources as *tainted*, dynamically propagate taint information along the execution of the target program, and detect attacks when tainted data are used for critical use, such as system calls.

Though DTPs are considered to have the potential to root out script injection attacks, existing systems still suffer from a trade-off between false positives and false negatives. To provide a solution the trade-off problem, Li et al. proposed a technique called ***SWIFT*** [1]. SWIFT takes a completely different approach to the conventional DTP systems. SWIFT tracks the memory accesses of the target program, detects pairs of interleaving read and write string accesses from the address trace, and propagates taint information from the read string to the written string. As will be explained in Section 3, SWIFT is free from the trade-off

that existing platform DTPs suffer from.

Although the basic idea of SWIFT is quite promising, Li et al. [1] only showed a preliminary implementation on a simulator and failed to show advantage in accuracy over Raksha [2], which is one of the most sophisticated platform DTP systems. Then, the purpose of this paper is to overcome these weak points. The contribution of this paper is as follows:

- We implement SWIFT to PHP.
- We succeeded to show that SWIFT has better propagation accuracy than Raksha in real-world web applications.

Implementation on PHP Since SWIFT need only the address traces of target programs, it is language-independent, that is, it can be implemented both as a software module of interpreters for script languages and as a hardware module of processors.

Li et al. implemented SWIFT on an IA-32 emulator ***Bochs*** [4] to evaluate the taint propagation accuracy.

On the other hand, we implement SWIFT to PHP interpreter to put SWIFT into practical use. We selected PHP because it is the most widely used in server-side web applications.

New evaluation results Li et al. failed to show the advantage in accuracy over Raksha because the web applications they used in the evaluation were too basic to show the difference.

We found out that the trade-off that existing DTPs suffer from is mainly caused from table references. ***Base64*** is a typical example of such table references. Then, we tried real-world web applications with vulnerabilities of Base64, and found Raksha can't detect attacks exploiting such vulnerabilities while SWIFT can.

The rest of the paper is organized as follows. Section 2 reviews background knowledge of script injection attacks and DTPs. Section 3 describes two types of string operations using table references to explain why existing DTPs are subject to a trade-off between false positives and false negatives. In this section, we also describe SWIFT in detail, which is free from the trade-off problem. Section 4 explains how we implemented SWIFT to PHP. Section 5 shows the evaluation results with respect to propagation accuracy and performance overhead of the PHP implementation. Finally, Section 6 states our conclusions and future work.

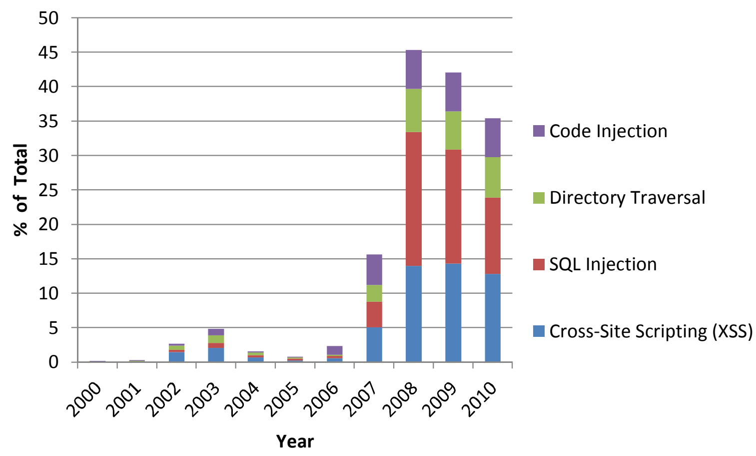

Figure 1.1: Increase in script injection attack

Chapter 2

Script Injection Attacks and Dynamic Taint Propagation

This section reviews background knowledge of script injection attacks and DTPs. Sections 2.1 and 2.2 explain SQL injection and its detection mechanism using DTPs. In Section 2.3, we explain command parsing, in conjuction with which SWIFT is assumed to be used. In Section 2.4, we discuss the problem with existing DTPs using Base64 as an example.

2.1 SQL injection

From cross-site scripting to SQL injection, attackers can use various techniques to attack web applications. This section uses SQL injection as an example to explain how script injection attacks occur.

SQL injection is one of the most common attacks. It allows an attacker to access sensitive information from a Web server's database. Figure 2.1 gives an example of SQL injection.

Assume that a web page displays the price of a product that a user has selected by typing its name into a text box. Figure 2.1(a) gives the associated PHP statement on the web page to achieve this. The string the user entered into the text box is stored in the variable `$name`. By concatenating `$name` and the constant string, the statement produces the SQL command `$cmd` that is sent to the SQL server.

Normally, a user enters a product name such as `ruby` for `$name`, resulting in the `$cmd` given in Figure 2.1(b). In this and the next subfigure, the substrings corresponding to `$name` are underlined. The database then returns the price of a

```
$cmd =
  "SELECT price FROM prod WHERE name=$name"
```

(a) PHP statement

```
$name:
  ruby

$cmd:
  SELECT price FROM prod WHERE name='\
  ruby'
```

(b) Non-attack string and resulting command

```
$name:
  dummy'; \
  UPDATE prod SET price=0 WHERE name='ruby

$cmd:
  SELECT price FROM prod WHERE name='\
  dummy'; \
  UPDATE prod SET price=0 WHERE name='ruby'
```

(c) Attack string and resulting command

Figure 2.1: Example of SQL injection

`ruby`.

If an attacker injects the string given in Figure 2.1(c) into `$name`, a different `$cmd` shown in the same figure is produced, which updates the database without the programmer's knowledge.

As seen in this example, a script injection attack is carried out by making the victim server interpret the string that includes attack code written in the scripting language. As for binary injection attacks, even if binary attack code is successfully injected, execution of the injected binary can easily be prohibited, e.g., using an NX bit. As for script injection attacks, however, interpretation of an injected script itself cannot be prevented because this is the key benefit of using scripting languages. This is the main difficulty in detecting script injection attack .

2.2 DTPs to Script Injection Attacks

The use of DTPs is a promising technique for detecting script injection attacks. The idea behind a DTP is to tag data from untrusted sources as tainted, for example, data from network I/O, user input, or read from any untrusted device. The tags are propagated during program execution. If tainted data is used in an unsafe way, such as a system call or a SQL command, attacks are detected.

The original inspiration behind the concept of DTPs was provided by the taint mode of Perl [5]. Since then, this kind of technique has been supported in various programming languages, such as PHP [6, 7, 8], Ruby [9], Java [10, 11], and C [12, 13] as well as its descendants. This language-level support is referred to as *language DTP*s. On the other hand, Suh et al. first applied the Perl taint mode to a processor to detect injection attacks to binary code, and called it DIFT (Dynamic Information Flow Tracking) [14]. We refer to such techniques on processors as *platform DTP*s. Although the purpose of platform DTPs was to detect binary injection attacks and platform DTPs were affected by interpreting noise, Dalton et al. pointed out that they could also detect script injection attacks and could provide the same level of accuracy as language DTPs [2].

Interpreting noise is defined as instructions executed only for the sake of interpreting scripts, and which provide no help in information flow tracking. Tens of native instructions are needed to interpret just a single instruction of an intermediate scripting language. These instructions are not directly related to information flow tracking, and thus, for detecting script injection attacks, they behave as noise.

In the example of SQL injection in Section 2.1, the tainted substring is underlined. Because SQL commands, such as *UPDATE* and *SET*, or field and table

names, such as *price* and *prod* are tainted, SQL injection can be detected.

In the next section, we explain command parsing, in conjuction with which SWIFT is assumed to be used.

2.3 Command Parsing

Su et al. showed that SQL injection can always be perfectly detected as long as the SQL syntax is known and the substrings are correctly detected as *trusted* or *untrusted* [15].

As in the example of SQL injection given in Section 2.1, the command parser of the SQL server knows which substrings must be trusted and which may be untrusted. Specifically, keywords, such as *UPDATE* and *SET*, or field and table names, such as *price* and *prod*, must be trusted, whereas arguments such as `ruby` could be untrusted. If the parser knows that the substring of `$cmd` corresponding to `$name`, i.e., the underlined data in the SQL injection, is untrusted, the parser can easily distinguish whether `$cmd` is an attack or not.

This command parsing can also be applied to any commands arising from web applications. In general, data from untrusted sources should not specify the names of the system resources, but may specify their contents. The names of system resources include file names, command names, or field and table names of databases.

Used with command parsing, therefore, it is not rational to let DTPs decide which substring is untrusted based on its own judgment. DTPs should always leave the substring corresponding to `$name` tainted even in non-attack cases. In the example in Section 2.1, even if `ruby` is left tainted, the parser can distinguish it from actual attack code. Academic researchers have had appropriate command parsing to prevent many kinds of script injection attacks [16] [13].

In the next section, we use Base64 as an example to explain the problem with existing DTPs.

2.4 Problems with Existing DTPs

Some web applications use Base64 to obfuscate sensitive input. For example, the code shown in Figure 2.2(a) is found in Cubecart 3.0.3

After the *base64_decode()*, `$redir` is not sanitized, and this could lead to a cross-site scripting attack.

```
$redir =
  base64_decode($_GET['redir']);
```

(a) PHP statement

```
http://[victim]cc3/index.php?act=login&
redir=L3NpdGUvZGVtby9jYzMvaW5kZXgucGhwP
2FjdD12aWV3RG9jJmFtcDtkb2NJZD0x
```

(b) Attack code

```
$redir:
  /site/demo/cc3/index.php?act=viewDoc&docId=1
```

(c) XSS code

Figure 2.2: Example of Base64 vulnerability

For example, if an attacker creates and inputs a specially crafted URL in Figure 2.2(b), `$redir` in Figure 2.2(c) is generated after the *base64_decode* function. When the code is executed, cross-site scripting occurs.

Existing DTPs do not propagate tainted information through Base64, and therefore they can not detect the cross-site scripting mentioned above. In the rest of this section, we explain why existing DTPs do not propagate tainted information through Base64.

The Base64 encoding procedure is as follows:

1. 3 uncoded bytes (8*3=24bits) are converted into 4 numbers (6*4=24bits)

2. 4 numbers are converted to their corresponding values using a conversion table

The Base64 decoding procedure is the reverse of the above process. The key point is that Base64 uses table reference for conversion. In general, a table reference is as follows:

```
$ostr = $table[$istr];
```

We can regard a table reference as safe in the usual cases, but if it is used for conversion, it is unsafe. In propagating tainted information, a table reference falls into the trade-off between false positives and false negatives. If we regard table references as safe, tainted information is not propagated from `$istr` to `$ostr`,

and this causes a security hole. On the contrary, if we regard table references as unsafe, tainted information is propagated from `$istr` to `$ostr`, and this results in plenty of false positives. Most existing DTPs select the former approach, so they do not propagate tainted information through Base64.

Chapter 3

SWIFT

This section explains SWIFT [1]. SWIFT provides much higher accuracy for detecting script injection attacks than conventional DTPs.

First, we consider the essence of DTPs in Section 3.1. Then, we describe two types of string operations on table references, namely, loop-in-select and select-in-loop, in Section 3.2. Finally, we present a detailed algorithm for SWIFT in Section 3.3.

3.1 Radio Button and Text Box String Operations

Radio buttons and text boxes are two representative user interfaces commonly found on web pages. Radio buttons are used to choose one of several options, while text boxes are used to obtain arbitrary strings.

As described in Section 2.1, text boxes are unsafe to injection attacks and the programmer must carefully check the string entered through a text box. On the other hand, radio buttons are considerably safer. Since the options of a radio button are provided by the programmer, the string that the user chooses is under control of the programmer, and it is practically impossible for attackers to attack through a radio button.

The string operations in web applications can also be divided into radio buttons and text boxes. In an application, strings travel from the input to the output, experiencing one or more radio button and/or text box operations. A substring of a string can be considered to be under the control of the programmer if it experiences at least one radio button operation en route.

Therefore, what DTPs should do is to identify whether the operation that a

```
$table['0'] = "ruby";
$table['1'] = "sapphire";
/* ...  */
  $ostr = $table[$i];
```

(a) Sample code for radio button string operation: safe

```
$table['a'] = 'A';
$table['b'] = 'B';
/* ...  */
for ($i = 0; $i < strlen($istr); $i++)
  $ostr[$i] = $table[$istr[$i]];
```

(b) Sample code for text box string operation: unsafe

Figure 3.1: Two types of string operations

substring undergoes is a radio button or text box, and to propagate tainted information from the input string to the output string only if the operation is detected to be a text box.

Note that this will propagate tainted information to the output even if it is not an attack. This is acceptable because SWIFT is assumed to be used in conjuction with command parsing described in Section 2.3. The parser can then detect that it is not an attack even if a substring such as `ruby` is tainted.

3.2 Loop-in-Select and Select-in-Loop Structures for Table References

In Section 2.4, we mentioned that table references fall into the trade-off between false positives and false negatives. In this section, we discuss in detail why this is the case.

Figure 3.1 shows that string operations using table references can be classified into two types: radio buttons and text boxes. Figure 3.1(a) shows sample code of radio button string operations. In this code, there is a table containing strings like `ruby` and `sapphire`. The string is assigned to `$ostr` by referencing the table. It is assumed that `$i` is specified by the user and is tainted. Figure 3.1(b) shows sample code of text box string operations. In this code, there is a table containing capital letters corresponding to each index. An arbitrary string is stored in `$istr` and the string is converted from lowercase to uppercase by referencing the table. As in Figure 3.1(a), it is assumed that `$istr` is specified by the user and is

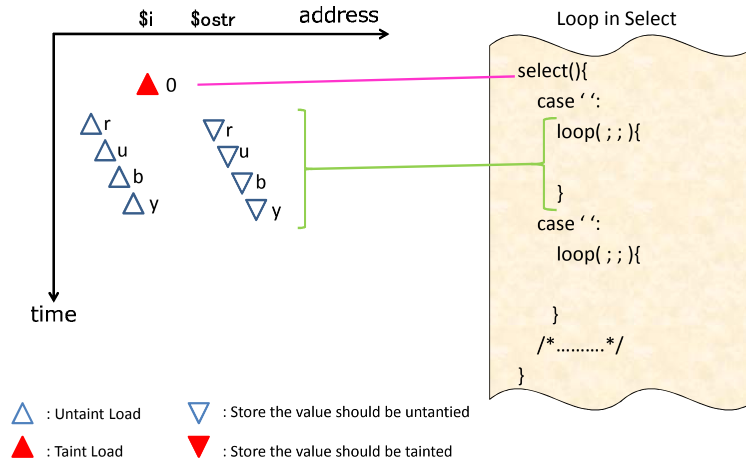

Figure 3.2: Loop-in-Select structure: safe

tainted.

The string operation in Figure 3.1(a) is safe because the string that the user chooses is necessarily under the control of the programmer. Thus it is practically impossible for attackers to attack through this operation. On the other hand, the string operation in Figure 3.1(b) is unsafe because the user can control the output and obtain an arbitrary string. It is thus, possible to attack through this operation. This string operation includes a string copy and all kinds of string conversions such as case or code conversions.

The string operations in Figure 3.1(a) and Figure 3.1(b) are almost the same except that the table reference in the latter unsafe string operation is used in the *for* statement. We can distinguish these two types by focusing on the address trace. Figure 3.2 and Figure 3.3 show the address traces of these string operations. In these figures, the x- and y-axes indicate the address and time, respectively. There are four types of triangles. Upward triangles △ and ▲ indicate load instructions of untainted and tainted data, respectively. Downward triangles ▽ and ▼ indicate store instructions whose store values that should be untainted and tainted, respectively. The load/store instructions that do not relate to DTP are not drawn in these figures.

Figure 3.2 corresponds to the sample code of the safe string operation shown in Figure 3.1(a). The first tainted load indicates the load of input variable `$i`. In this case, the value of `$i` is `'0'`, and then the constant string `"ruby"` is copied to the output variable `$ostr`. Figure 3.3 corresponds to the sample code of the

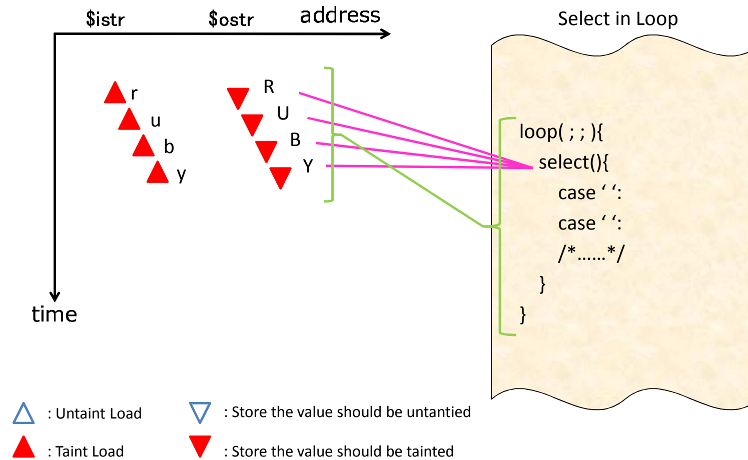

Figure 3.3: Select-in-Loop structure: unsafe

unsafe string operation shown in Figure 3.1(b). The tainted input string "ruby" is converted to "RUBY".

In both figures, the load instructions of the string read and the store instructions of the string write appear in an interleaved fashion. The obvious difference is that the loads are untainted in the safe string operation, whereas they are tainted in the unsafe string operation. If we regard the table reference as a "select" and the interleaving read/write as a "loop", we can see that the safe string operation can be represented by the ***loop-in-select*** structure in Figure 3.2. We can also see that the unsafe string operation is depicted by the ***select-in-loop*** structure in Figure 3.3. Existing DTPs do not pay attention to non-local structures such as loop-in-select and select-in-loop, and are therefore prone to the table reference trade-off.

3.3 Algorithm of SWIFT

In this section, we explain the algorithm of SWIFT in detail.

3.3.1 Overview

SWIFT is a proper method for distinguishing loop-in-select and select-in-loop structures. Unlike existing DTPs, SWIFT does not track information flow instruction by instruction. Instead, SWIFT only observes the address traces of executed

load/store instructions and detects select-in-loop string operations.

SWIFT detects sequential memory accesses as string accesses. Moreover, SWIFT detects an interleaving string read and write as a string operation. If the read string is tainted, SWIFT detects this string operation as a select-in-loop string operation and propagates the tainted information from the read string to the write string.

3.3.2 Select-in-Loop String Operation Detection

Streams and Interleaving Pairs

A ***read stream*** is a sequence of read accesses to a string, and a read access in a read stream is referred to as a ***stream read***. Likewise, a ***write stream*** is a sequence of write accesses to a string, and a write access in a write stream is referred to as a ***stream write***.

The purpose of the algorithm is to detect an ***interleaving pair*** of a read stream and a write stream. Figure 3.4 shows an example of an interleaving pair. This figure shows an address trace of *base64_encode*. In an interleaving pair, the stream reads and writes appear alternatively. The read stream is divided into multiple read ***substreams*** through occurrences of the stream writes, and vice versa. Each of the read/write substreams contains one or more stream reads/writes. A read/ write access in the read/write stream of an interleaving pair is referred to as an ***interleaving-stream read/write***.

Tables

Two tables are used to detect read and write streams. Each of the entries in the read/write stream tables is allocated to a stream.

An entry in the tables has the following fields:

- *start* The start address of the stream.
- *next* The predicted next address of the stream.
- *n_access* The current number of accesses in the stream.
- *n_substrm* The current number of substreams in the stream.
- *switched* A flag to calculate *n_substrm*.

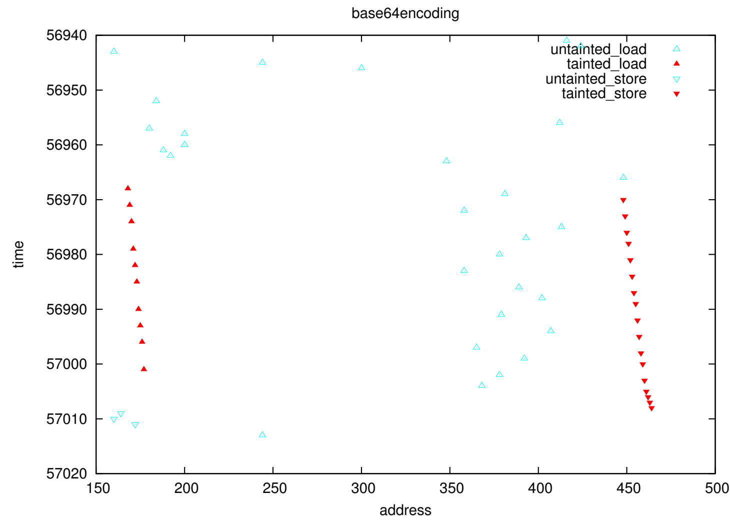

Figure 3.4: Address trace of *base64_encode*

Stream Read/Write Detection

On a read access to *addr*, the *next* value of all the entries in the read table is compared to *addr*. If there is no match, a new entry is created, *start*, *next*, *n_access* are initialized to *addr*, the address next to *add*, and one. If there is a match, *n_access* is incremented and *next* is advanced for the future access. An entry with *n_access* greater than a threshold is recognized as a read stream. In other words, if *addr* matches the *next* and *n_access* is greater than a threshold, the read access is detected as a stream read. The same holds true for the write table and write accesses.

Interleaving Stream Read/Write Detection

When a stream write is detected, the *switched* flags of all the entries in the ***read*** (not write) table are set. Thereafter, a read access of a stream is detected as the first access to a new substream because *switched* is set. Then, *n_substrm* is incremented, and *switched* is reset for the possible second access in the same substream. An entry with *n_substrm* greater than a threshold is detected as the read stream of an interleaving pair. In other words, if *addr* matches *next* and *n_substrm* is greater than a threshold, the read access is detected as an interleaving stream read. The same holds true for the write table and write accesses.

3.3.3 Propagation and Backtracking

Every time a stream read is detected, the taintedness of the read is stored in the *taintedness* variable. Then, when an interleaving stream write is detected, the taintedness of the written word is set to the value of *taintedness*.

When the detector detects streams, the same number of accesses as the threshold have already been performed. Thus, ***backtracking*** is needed, that is, these written characters should also be tainted. The *start* field of the entry is primarily used to locate the start address of the stream.

Chapter 4

Implementation of SWIFT to PHP

We implement SWIFT to PHP interpreter to put SWIFT into practical use. We selected PHP because it is the most widely used in server-side web applications.

First we give an overview of our implementation in Section 4.1. Thereafter, we describe how to cope with native functions in Section 4.2. In Section 4.3, we explain the PHP interpreter. Finally, we explain in detail how to acquire memory addresses from the source code of the interpreter in Section 4.4.

4.1 Overview

Figure 4.1 shows a general overview of our implementation. A PHP script is compiled to intermediate code by a runtime compiler and is executed by an executor.

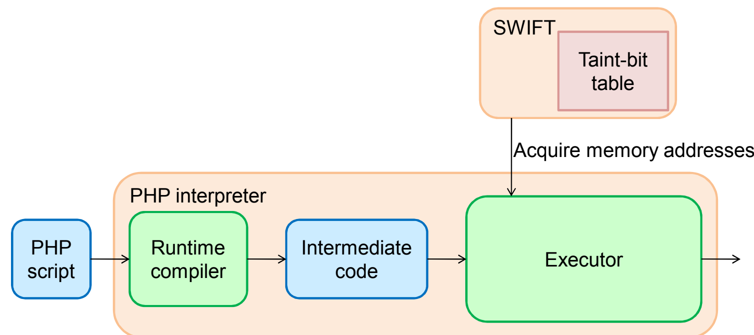

Figure 4.1: General overview of our implementation

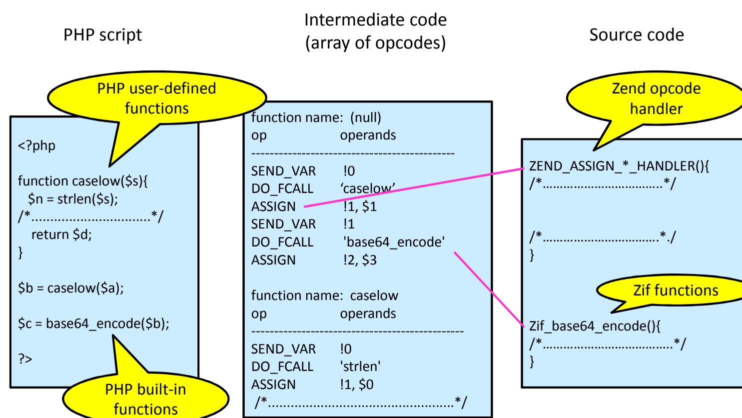

Figure 4.2: Relationship among script, intermediate code and source code

The PHP interpreter is written in C. SWIFT's engine has a hash table of taint-bits whose access keys are memory addresses. SWIFT is written in C++.

Since SWIFT focuses only on address traces of the program execution, we must obtain memory addresses corresponding to the load/store instructions from the PHP interpreter. In this research, we insert hook functions in the source code of the executor. As described in subsequent sections, hook functions can be automatically inserted in a fairly simple algorithm using our Perl program. Because we utilize information from the source code, only the memory addresses of strings can be acquired. If we exactly obtain memory addresses of the strings, it is possible to propagate tainted information correctly using the SWIFT algorithm described in Section 3.3. Note that SWIFT can work even if all memory addresses are obtained, that is, including addresses other than those of strings.

4.2 Coping with Native Functions

Parts of an interpreter are generally implemented as native functions.

In platform DTPs, it is difficult to propagate tainted information through an interpreter because there is plenty of interpreting noise as mentioned in section 2.2. Conversely, it is easy to do this through native functions because there is no influence of interpreting noise.

In language DTPs, however, the ease of taint propagation mentioned above is reversed. It is easy to propagate tainted information through an interpreter be-

cause we can utilize the interpreter's information, whereas it is difficult to do this through native functions because we can not utilize the interpreter's information.

As described in detail in subsequent sections, it is fairly easy for us to obtain memory addresses from the source code of the interpreter, because we can identify the memory accesses to strings by merely focusing on the macros.

On the other hand, it is generally difficult to implement DTPs to native functions, and it also seems to be difficult to obtain the memory addresses of strings from the source code of native functions. However, we are able to insert hook functions automatically in a simple algorithm using our Perl program.

All variables are stored in a data structure called *zval* in the PHP interpreter. The string data of *zval* are passed to the native functions in a uniform way in the form of char pointers. In native functions, strings are treated as char pointers. After processing of the native functions, the string data are passed to *zval* in a certain way. This is why memory addresses can be acquired easily by focusing only on char pointers in the source code of the native functions.

If built-in C library functions are used, we do not know exactly how load/store instructions are executed internally. However, if built-in C library functions, such as *memcpy* and *strcpy*, are executed, we must obtain memory addresses because these instructions operate as text box operations and move the location of strings by interleaving load/store accesses. If the source code of these is available, memory addresses of the load/store instructions can be obtained. In practice, however, the source code may not be available.

To obtain memory addresses and to propagate tainted information correctly through built-in C library functions, we take advantage of source and destination addresses, and the number of moved bytes. For example, if *memcpy* is used, the data pointed to by the second argument is copied to the memory block pointed to by the first argument where the number of bytes to copy is specified by the third argument. Therefore, the data pointed to by the second argument can be regarded as a read string, with size equal to the number of bytes specified in the third argument, while the data pointed to by the first argument can be regarded as a write string, with the same size as the source string. Tainted information of the read string is directly propagated to the write string.

We also know the memory blocks of read and write strings for other built-in C library functions, so tantied information can be propagated correctly.

```
struct _zval_struct {
    zvalue_value value; /* value */
    zend_uint refcount__gc;
    zend_uchar type;
    zend_uchar is_ref__gc;
}zval ;
```

Figure 4.3: Zval structure

4.3 PHP Interpreter internals

In this section, we discuss the knowledge of the PHP interpreter internals needed to describe our implementation details in the next section.

4.3.1 Relationship among Script, Opcode and Source code

Figure 4.2 shows the relationship among the script, intermediate code and source code of the PHP interpreter. On the left of Figure 4.2 is a sample PHP script. The intermediate code in the middle has been dumped using vld [17].

The intermediate code is an ordered array (an *op array*) of instructions (known as *opcodes*) [18], such as *DO_FCALL* and *ASSIGN*. We call the source code of each opcodes a ***zend opcode handler***.

The sample script uses *base64_encode*, which is a *PHP built-in function*. We call the source code of each PHP built-in function a ***zif function***. Zif functions correspond to native functions.

The sample script also uses *caselow*, which is a *PHP user-defined function*. In the bottom half of the intermediate code, the PHP user-defined functions are considered op arrays as well, as if they were miniature scripts.

Therefore, we have only to acquire memory addresses from ***zend opcode handlers*** and ***zif functions***.

4.3.2 Variable Management

In PHP, all variables are *zvals*. Figure 4.3 and Figure 4.4 show the C structure of a zval and its complementary data container, respectively.

To access various types of data, macros in source code of interpreters, which take zvals as their arguments, can be used. For instance, if we wished to extract the string buffer (*char* val* in Figure 4.4) for *zval*, *zval** and *zval***, we would

```
typedef union _zvalue_value {
    long lval;
    double dval;
    struct {
        char *val; /* string value */
        int len;
    } str;
    HashTable *ht;
    zend_object_value obj;
} zvalue_value;
```

Figure 4.4: Data container for zval

```
#define Z_STRVAL(zval)          (zval).value.str.val
#define Z_STRVAL_P(zval_p)      Z_STRVAL(*zval_p)
#define Z_STRVAL_PP(zval_pp)    Z_STRVAL(**zval_pp)
```

Figure 4.5: zval-to-C data type conversion macros

use *Z_STRVAL*, *Z_STRVAL_P* and *Z_STRVAL_PP*, respectively, as defined in the source code(Figure 4.5).

4.3.3 Memory Management

The PHP interpreter uses its own internal memory-management wrapper functions as given in Table 4.1. In the following description, we regard memory-management wrapper functions as built-in C library functions. In particular, we focus on *erealloc* and *estrndup*.

4.3.4 Declaration of zif functions

Figure 4.6 shows the declaration of zif functions using *zif_base64_encode* as an example. In a zif function, *zend_parse_parateters* is used to extract the variables passed into a zif function from the zvals. The third argument "s" specifies the string type of data. The next argument, &str, is a reference to the C variable that fills out with the value of the argument. As a result, the string data of the zval is treated as the char pointer in zif functions—in this case "str".

Table 4.1: Memory-management Wrapper Functions

Function	Usage
void *emalloc(size_t size)	malloc() replacement
void efree(void *ptr)	free() replacement
void *erealloc(void *ptr, size_t size)	realloc() replacement
char *estrndup(char *str)	strndup() replacement

```
01    PHP_FUNCTION(base64_encode) //zif_base64_encode
02    {
03      char *str;
04      unsigned char *result;
05      int str_len, ret_length;
06
07      if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC,
08      "s", &str, &str_len) == FAILURE) {
09          return;
10      }
11      result = php_base64_encode((unsigned char*)str, str_len, &ret_length);
12      if (result != NULL) {
13          RETVAL_STRINGL((char*)result, ret_length, 0);
12      } else {
13          RETURN_FALSE;
14      }
15    }
```

Figure 4.6: Declaration of zif functions

4.4 Acquisition of Memory Addresses

In this section, we explain how to acquire memory addresses. Only memory addresses of strings can be acquired because we can utilize information from the interpreter's source code. As explained in Section 4.3.1, we only need to acquire memory addresses from zend opcode handlers and zif functions.

4.4.1 Zend Opcode Handler

The string access opcodes are the following:

- ASSIGN, ASSIGN_DIM
- ADD_CHAR, ADD_STRING, ADD_VAR
- BW_AND, BW_NOT, BW_OR, BW_XOR
- CONCAT
- POST_INC, POST_INC_OBJ

```
1    swift_load(&Z_STRVAL_P(T->str_offset.str)[T->str_offset.offset],1);
2    Z_STRVAL_P(T->str_offset.str)[T->str_offset.offset] = Z_STRVAL(tmp)[0];
3    swift_store(&Z_STRVAL(tmp)[0],1);
```

(a) The macros access an element of a string using subscripts

```
1    memcpy(Z_STRVAL_P(result), Z_STRVAL_P(op1), Z_STRLEN_P(op1));
2    swift_memcpy(Z_STRVAL_P(result), Z_STRVAL_P(op1), Z_STRLEN_P(op1));
```

(b) built-in C library functions take the macros as their arguments

Figure 4.7: Acquisition of memory addresses from zend opcode handler

- PRE_INC, PRE_INC_OBJ

We focus on the zend opcode handlers corresponding to these opcodes. In a zend opcode handler, string access is done using the macros mentioned in Section 4.3.2, and thus we can recognize a string access. We insert hook functions in the following two cases.

The first case occurs when the macros access an element of a string using subscripts. For example, assume the source code given in line 2 in Figure 4.7(a). If the string access is on the right of the assignment operator, we regard it as a load access. On the other hand, if the string access is on the left of the assignment operator, we regard it as a store access. In this code, both load and store accesses occur. We use *swift_load* and *swift_store* as hook functions to obtain memory addresses in lines 1 and 3 in Figure 4.7(a). The first argument of these functions is the memory address of the load/store access while the second argument is the byte size of the load/store access. In our implementation, the second argument is always 1.

The second case occurs when built-in C library functions take macros as their arguments. The functions to which we should pay attention are the following: *estrndup*, *erealloc*, *memcpy* and *memmove*.

For example, assume the source code as in line 1 in Figure 4.7(b). In this case, *Z_STRVAL_P(op1)* corresponds to a read string while *Z_STRVAL_P(result)* corresponds to a write string, and thus, we should obtain the memory addresses of *Z_STRVAL_P(op1)* and *Z_STRVAL_P(result)*. We use *swift_memcpy* as a hook function as in line 2 in Figure 4.7(b). The first and second arguments are, respectively, a pointer to the destination and a pointer to the source. The third argument

is the number of bytes to copy.

4.4.2 Zif Functions

The following functions constitute zif functions for which we must obtain memory addresses.

- String Functions(e.g., *htmlentities*)
- POSIX Regular Expression Functions(e.g., *ereg_replace*)
- Perl Compatible Regular Expression Functions (e.g., *preg_replace*)
- URL Functions(e.g., *base64_encode*)
- Multibyte String Functions(e.g., *mb_eregi_replace*)

Because only char pointers are passed to zif functions, we can not insert hook functions focusing on the macros. As mentioned in Section 4.2, hook functions can be inserted automatically into zif functions based on a simple algorithm in our Perl program. In this algorithm, we need only consider char pointers. As in zend opcode handlers, hook functions are inserted in the following two cases.

The first case occurs when a char pointer accesses its element using an asterisk or subscript. If the string access is on the right of the assignment operator, we regard it as a load access. On the other hand, if the string access is on the left of the assignment operator, we regard it as a store access. In the source codes for *base64_encode*, for example, we obtain memory addresses as in Figure 4.8. As described in Section 2.4, Base64 uses table references for conversion. In line 10, *base64_table* is used as the conversion table. When line 10 is executed, *current[0]* is loaded and **p* is stored. As a result, we obtain the memory addresses of *current[0]* and *p* as load and store accesses in lines 9 and 12, respectively. Lines 11 and 13 are required to correct the memory addresses pointed to by *p*.

The second case occurs when a built-in C library function takes a char pointer as its argument. According to our investigation, the functions to which we should pay attention are the following: *estrndup*, *erealloc*, *memcpy*, *memmove*, *strcpy*, *strlcpy*, *strncat*, and *strlcat*. *swift_memcpy* is also used to obtain memory addresses of these built-in C library functions.

```
01  PHPAPI unsigned char *php_base64_encode
02  (const unsigned char *str, int length, int * ret_length)
03  {
04    const unsigned char *current = str;
05    unsigned char *p;
06    unsigned char *result;
07    /* ... */
08    while (length <2) {
09       swift_load((char*)&current[0],1);
10       *p++ = base64_table[current[0] >>2];
11       p--;
12       swift_store((char*)&p,1);
13       p++;
14    /* ... */
15  }
```

Figure 4.8: Acquisition of memory addresses from zif function

Chapter 5

Evaluation

We evaluated the taint propagation accuracy of PHP-SWIFT compared with Raksha and PHP-taint.

Raksha [2] is one of the most sophisticated platform DTPs. Since Raksha does not by default track address flows, we implemented algorithms with and without address flow tracking, which we refer to as Raksha−a and Raksha+a, respectively.

Regarding PHP-taint, we used the PHP-taint 20080622 package [6]. PHP-taint was the first implementation of taint support for PHP released in November 2007.

In addition, we also evaluated the performance overhead of PHP-SWIFT.

In Section 5.1, we explain our evaluation method. Sections 5.2 and 5.3 show the experimental results of the taint propagation accuracy and performance overhead, respectively.

5.1 Evaluation Method

5.1.1 Environment

We used the IA-32(x86) emulator Bochs 2.3.5 [4] to implement Raksha. We added per-byte taint bits and their propagation to Bochs. Although the actual proposal for Raksha was implemented at word granularity in taint tracking and an implementation of Raksha at byte granularity increases the precision of taint tracking, in order to test and verify the taint propagation precision fairly, our implementation of Raksha supported byte granularity.

Our measurements were taken on a 3.2 GHz Intel Core i7 machine with 6 GB RAM. Table 5.1 gives the details of our evaluation environment.

Table 5.1: Evaluation environment

	PHP-SWIFT	PHP-taint	Raksha
Host OS	Windows Vista x64		
Emulator	VMware Player 3.1.4		Bochs 2.3.5
Guest OS	Ubuntu 10.04		Red Hat Linux 8.0
Web Server	Apache 2.2.14		Apache 2.2.11
SQL Server	MySQL 5.1.37		MySQL 4.1.22
PHP	modified PHP-5.3.1	PHP-taint 20080622 package	PHP-5.2.5

5.1.2 Methodology

PHP-SWIFT and PHP-taint automatically taint data from outside the PHP scripts. As for Raksha, data from the network are manually tainted when the data are read by Apache because PHP interpreter is installed as an Apache module.

To evaluate the taint propagation accuracy of PHP-SWIFT and Raksha, we checked which substrings of the output strings were tainted when the target programs called sensitive functions. As for PHP-taint, we merely recorded the security exceptions it raised.

To evaluate the performance overhead of PHP-SWIFT, we inserted a *microtime* function, which is a PHP built-in function, at the beginning and end of the scripts and measured the execution time of the interpreted scripts.

5.2 Taint Propagation Accuracy

To evaluate the taint propagation accuracy of PHP-SWIFT, PHP-taint, and Raksha, we executed typical string operations and a wide range of common script injection attacks on the real-world web applications.

5.2.1 String Operations

Table 5.2 summarizes the results of the basic string operations. PHP-SWIFT correctly propagates tainted information for all the operations, whereas Raksha and PHP-taint do not. The string operations include string copies and case and code conversions, which are commonly used in web applications. Operations(2) to (7) are PHP built-in functions, and thus they are written in C.

(1) concatenation, (2) *substr()*, and (3) *ereg_replace()* each execute a string copy at the end of the operation, and all the models propagate taint correctly.

(4) *ereg()* is a regular expression match, and all the models untaint the scalar result.

Table 5.2: Results of string operation

	Operation	PHP-SWIFT		PHP-taint		Raksha−a		Raksha+a	
		FN	FP	FN	FP	FN	FP	FN	FP
(1)	concatenation								
(2)	*substr()*								
(3)	*ereg_replace()*								
(4)	*ereg()*								
(5)	*strtoupper/tolower()*					√			
(6)	*urlencdoe/decode()*			√		√		√	
(7)	*base64_encode/decode()*			√		√			
(8)	untaint table								√
(9)	taint table			√					√
(10)	toupper (switch-statement)			√		√		√	

FN : false negative FP : false positive

(5) *strtoupper/strtolower()* are case conversions. Raksha+a taints the output, whereas Raksha−a does not. This is because the functions use a translation table.

(6)*urlencode/urldecode()* and (7)*base64_ encode/base64_ decode()* carry out encode and decode operations. As a result, PHP-taint untaints the outputs of all these functions. This may be a matter of policy. Since these code conversions include address and control flows, they are difficult for Raksha to deal with. Although Raksha+a tracks address flows, both Raksha+a and Raksha−a produce false negatives in (6) *urlencode/urldecode()* because they do not track control flows.

(8) untaint table and (9) taint table retrieve values from tables with taint keys, and store untainted and tainted values, respectively. Since PHP-taint regards the values from tables as safe, it results in false negative in operation (9), taint table. On the other hand, PHP-SWIFT and Raksha can track the flow between the input and the output values through a table. SWIFT and Raksha taint the contents of the table when the tainted input is copied to it, and also taint the output of the table when it is copied from the tainted contents. Raksha+a, however, also propagates the taint from the index to the output because it tracks address flows, resulting in false positives.

Operation (10) is an uppercase conversion. Though the function is the same as (5) *strtoupper()*, it is written in PHP like Figure 3.1(b). Operation (10) is written with a switch statement construction. Raksha−a and Raksha+a produced false negatives for switch statement because they untainted the input when it was compared. PHP-SWIFT produced no false positives or negatives because PHP-SWIFT can correctly propagate tainted information for all the operations. Therefore, even if programmers use operations such as these as the input arguments for applications, PHP-SWIFT can still provide high precision.

```
http://[target]/qwiki/index.php?page=
../_config.php%00
```

(a) Attack code

```
data/../_config.php%00
```

(b) Path argument

Figure 5.1: Qwikiwiki 1.4.1 vulnerability

5.2.2 Real-World Web Applications

Web Applications with Known Vulnerabilities

We executed seven web applications with known vulnerabilities written in PHP. The applications are phpSysInfo 2.3, QwikiWiki 1.4.1, phpBB 2.0.8, PHP-Nuke 7.5, Cubecart 3.0.3 and PHP-Nuke 7.1. In choosing the web applications, we selected applications whose specific exploit codes could be found on the web. These applications use various input variables as arguments without validation or even with string operations applied to them. We conducted script injection attacks such as cross-site scripting (XSS), SQL injection, and directory traversal according to the exploit code. As summarized in Table 5.3, PHP-SWIFT produced no false positives or negatives. However, Raksha−a caused several false negatives and Raksha+a resulted in plenty of false positives. PHP-taint also produced false negatives.

Cubecart 3.0.3 and PHP-Nuke 7.1 use Base64 in a risky way. As described in Section 2.4, to exploit these vulnerabilities, attackers should encode their attack code using Base64, and use this base64 encoded code to create a specially crafted URL. Since Raksha−a does not deal with address flows, it can not propagate tainted information correctly through Base64. Therefore, it produced false negatives in these applications. Raksha+a resulted in frequent false positives in all the web applications because it makes a great part of the address space tainted.

The vulnerabilities of other web applications are elementary.

In the rest of this section, we explain QwikiWiki 1.4.1 and PHP-Nuke 7.1 in detail.

QwikiWiki 1.4.1 QwikiWiki 1.4.1 has a directory traversal vulnerability in "index.php". It allows attackers to read arbitrary files via a .. (dot dot) and a %00 at the end of the filename in the page parameter.

```
$nukeuser =
  base64_decode($user);
```

(a) PHP statement

http://[target]/nuke71/modules.php?name=
Private_Messages&file=index&folder=
inbox&mode=read&p=1&user=eDpmb28nIFVO
SU9OIFNFTEVDVCAyLG51bGwsMSwxLG51bGwvKjox

(b) Attack code

```
$nukeuser:
  x:foo' UNION SELECT 2,null,1,1,null/*:1
```

(c) SQL injection code

Figure 5.2: PHP-Nuke 7.1 vulnerability

For example, assume the attack code is configured as in Figure 5.1(a). Then, it raises an *open()* system call with the string in Figure 5.1(b) as its path argument. This will open *_config.php*, which an attacker is not allowed to access. PHP-SWIFT successfully tainted the underlined substring.

QwikiWiki first stores the parameters such as *page* in a hash table, and then uses them from the table. Therefore, the same situation as in operation (9), taint table, in the previous section arises.

PHP-Nuke 7.1 PHP-Nuke 7.1 has a SQL injection vulnerability in ”modules.php”. When we checked the source code of ”modules.php”, we found the code as shown in Figure 5.2(a).

We can see the Base64 decoded global variable `$nukeuser`; however, the application does nothing to validate the variable. This means the variable `$nukeuser` can contain user supplied data without sanitization and this can lead to SQL injection.

For example, if an attacker inputs a specially crafted URL in Figure 5.2(b), `$nukeuser` in Figure 5.2(c) is generated after the *base64_decode* function. Concatenating the substring of `$nukeuser` and the constant strings provided by the programmer, produces the attack SQL query.

Thus, we can bypass the user-level authentication via this exploit.

Table 5.3: Results of web applications with known vulnerabilities

Program	Attack	PHP-SWIFT		PHP-taint		Raksha−a		Raksha+a	
		FN	FP	FN	FP	FN	FP	FN	FP
phpSysInfo 2.3	Cross-site scripting								√
QwikiWiki 1.4.1	Directory traversal			√					√
phpBB 2.0.8	Cross-site scripting			√					√
PHP-Nuke 7.5	SQL injection			√					√
CubeCart 3.0.3	Cross-site scripting			√		√			√
PHP-Nuke 7.1	Cross-site scripting			√		√			√
PHP-Nuke 7.1	SQL injection			√		√			√

FN false negative FP. false positive

Table 5.4: Results of Latest web applications

Program	Number of requests(valid requests)	FN	FP
Drupal 7.2	7(5)		
osCommerce 3.0.1	959(17)		
phpBB 3.0.8	180(28)		
WebCalendar 1.2.3	10(1)		
WordPress 3.3.1	523(272)		
XOOPS 2.2.0	348(24)		
Gallery 3.0.2	11(0)		

Latest Web Applications

To carry out further evaluations of PHP-SWIFT, we executed the latest famous web applications written in PHP. The applications are Drupal 7.2, osCommerce 3.0.1, phpBB 3.0.8, WebCalendar 1.2.3, WordPress 3.3.1, XOOPS 2.2.0, and Gallery 3.0.2. For each application, we assigned a value to each parameter of the GET and POST requests, and started each application. As shown in the second column of Table 5.4, valid requests were approximately 17% of all requests that were generated. As summarized in Table 5.4, PHP-SWIFT produced no false positives and negatives.

5.3 Execution Speed

Table 5.5 gives the performance overheads for web applications relative to an unmodified version of PHP. The table includes the execution time of unmodified PHP(column 2), the execution time of PHP-SWIFT(column 3), and the performance overhead(column 4). The overhead is between 6% and 95%, with an average of 55%. This overhead is greater than previous results reported by PHP-taint [6]. Optimization of the implementation is our future work.

Table 5.5: Performance overheads

Program	Attack	Unmodified PHP(ms)	PHP-SWIFT(ms)	Overhead
phpSysInfo 2.3	Cross-site scripting	1.10	1.16	6%
QwikiWiki 1.4.1	Directory traversal	9.25	18.08	95%
phpBB 2.0.8	Cross-site scripting	13.96	23.99	72%
PHP-Nuke 7.5	SQL injection	20.58	27.73	35%
CubeCart 3.0.3	Cross-site scripting	45.59	84.05	84%
PHP-Nuke 7.1	Cross-site scripting	25.27	38.66	53%
PHP-Nuke 7.1	SQL injection	21.31	29.04	36%

Chapter 6

Conclusion

In this paper, we implement SWIFT to PHP interpreter to put SWIFT into practical use. Moreover, we succeeded to show that SWIFT has better propagation accuracy than Raksha in real-world web applications.

SWIFT uses a completely different approach to that of conventional DTPs. To detect script injection attacks precisely, SWIFT observes memory accesses of the target program, detects string operations, and propagates tainted information through them. Since SWIFT only uses address traces of a program, it can be implemented both as a software module of interpreters for script languages and as a hardware module of processors.

Having implemented SWIFT to PHP, we compared its accuracy with that of PHP-taint and Raksha and evaluated the performance overhead. PHP-SWIFT can correctly propagate tainted information for typical string operations and real-world web applications with known vulnerabilities, whereas PHP-taint and Raksha cannot. The average performance overhead is 55%.

In this research, we created a naive implementation of SWIFT to PHP. Clearly, it is necessary to consider optimizations for the implementation and reevaluate the system. This is our future work.

Another aim of our future work is to carry out further evaluations using real-world web applications without known vulnerabilities to verify that no false positives are produced.

We intend distributing PHP-SWIFT in the near future.

Bibliography

[1] K. Li, R. Shioya, M. Goshima, and S. Sakai, "String-wise information flow tracking against script injection attacks," in *IEEE Int'l Symp. on Pacific Rim Dependable Computing (PRDC 2009)*, 2009, pp. 169–176.

[2] M. Dalton, H. Kannan, and C. Kozyrakis, "Raksha: A flexible information flow architecture for software securit," in *34th Int'l Symp. on Computer Architecture*, 2007, pp. 482–493.

[3] NIST, "National vulnerability database." [Online]. Available: http://web.nvd.nist.gov/view/vuln/statistics

[4] "Bochs: the open source IA-32 emulation project." [Online]. Available: http://bochs.sourceforge.net

[5] J. Allen, "Perl version 5.8.8 documentation - perlsec," 2006. [Online]. Available: http://perldoc.perl.org/perlsec.pdf

[6] W. Venema, "Taint support for php," 2008. [Online]. Available: http://wiki.php.net/rfc/taint

[7] T. Pietraszek and C. Berghe, "Defending against injection attacks through context-sensitive string evaluation," in *8th Int'l Symp. on Recent Advances in Intrusion Detection*, 2005, pp. 124–145.

[8] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans, "Automatically hardening web applications using precise tainting," in *20th Int'l Information Security Conf.*, 2005.

[9] D. Thomas and A. Hunt, *Programming Ruby: The Pragmatic Programmer's Guide*. Addison-Wesley Pub (Sd), 2000.

[10] V. Haldar, D. Chandra, and M. Franz, "Dynamic taint propagation for java," in *21st Annual Computer Security Applications Conf.*, 2005.

[11] B. Livshits, M. Martin, and M. S. Lam, "SecuriFly: Runtime protection and recovery from web application vulnerabilities," in *Tech. Rep., Stanford Univ.*, 2006.

[12] W. Xu, S. Bhatkar, and R. Sekar, "Practical dynamic taint analysis for countering input validation attacks on web applications," in *Tech. Rep. SECLAB-05-04*, 2005.

[13] ——, "Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks," in *15th USENIX Security Conf.*, 2006, pp. 121–136.

[14] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," in *11th Int'l Conf. on Architectural Support for Programming Languages and Operating System*, 2004.

[15] Z. Su and G. Wassermann, "The essence of command injection attacks in web applications," in *33rd Symp. on Principles of Programming Languages*, 2006.

[16] M. Dalton, "The design and implementation of dynamic information flow tracking systems for software security," Ph.D. dissertation, Stanford University, 2009.

[17] "vld." [Online]. Available: http://pecl.php.net/package/vld

[18] G. Schlossnagle, *Advanced PHP Programming*. DEVELOPER'S LIBRARY, 2003.

Publications

International Conferences

1. Hiroshi Toi, Ryota Shioya, Masahiro Goshima, and Shuichi Sakai:
   Yet Another Taint Mode for PHP,
   IEEE Int'l Symp. on Pacific Rim Dependable Computing (PRDC) (2010).
   (poster).

Domestic Symposiums(with peer review)

1. Hiroshi Toi, Ryota Shioya, Masahiro Goshima, and Shuichi Sakai:
   Yet Another Taint Mode for PHP,
   Symp. on Advanced Computing Systems & Infrastructures (SACSIS), Vol. 2011, pp. 160-169 (2011). (in Japanese).

Oral Presentations

1. Hiroshi Toi, Ryota Shioya, Masahiro Goshima and Shuichi Sakai:
   Implementation and Evaluation of String-Wise Information Flow Tracking to PHP,
   IPSJ SIG Technical Report 2010-OS-115, No. 4 (2010). (in Japanese).

2. Hiroshi Toi, Ryota Shioya, Masahiro Goshima and Shuichi Sakai:
   Implementation of String-Wise Information Flow Tracking to PHP,

The 72th National Convention of IPSJ, pp. 3-623-3-624 (2010). (in Japanese).

Acknowledgements

I am deeply indebted to many people for their contributions towards this master's thesis.

I would like to express my gratitude to Professor Shuichi Sakai, who has been my supervisor during my master course period. He gave me constructive comments and warm encouragement.

My deepest appreciation goes to Associate Professor Masahiro Goshima for his enthusiastic guidance. He provided me with many helpful suggestions and invaluable advice.

I would like to thank Ryota Shioya for his insightful comments and suggestions during my fourth year undergraduate days and my first year graduate days.

Special thanks to the rest of laboratory members. They have supported me from various aspects during everyday life.

I want to thank Ms. Harumi Yagihara, Ms. Tamaki Hasebe and Ms. Tomoyo Ise for their dedications to make administration affairs run smoothly.

Finally, I would like to thank my family and friends in the university of tokyo and elsewhere for their support.