Yet Another Taint Mode for PHP

HIROSHI TOI,† RYOTA SHIOYA,†† MASAHIRO GOSHIMA†
and SHUICHI SAKAI †

Nowadays, security of web applications faces a threat of script injection attacks, such as cross-site scripting (XSS), or SQL injection. DTP (Dynamic Taint Propagation) has been established as powerful techniques to detect script injection attacks. However, current DTP systems still suffer from trade-off between false positives and negatives, because these systems propagate tainted information from source operands to destination operands. We proposed SWIFT, which traces memory accesses of a program execution, detects string access and distinguishes string operations from other memory accesses, and propagates tainted information under string operations. This makes SWIFT provide a better accuracy on detection of script injection attacks than the current DTP systems. Since SWIFT only concentrates on address traces of a target program, it can be implemented both on interpreters of script languages and on hardware mechanisms of processors. In this paper, We implemented SWIFT to PHP, executed typical string operations and made injection attacks to some real-world web applications with known vulnerabilities. As a result of our experiments, SWIFT on PHP shows a high precision. Moreover, we evaluated the performance overhead. The average performance overhead is 55%.

1. Introduction

With an increase in web applications, attacks exploiting vulnerabilities of them have also been increasing. The attackers exploit various security vulnerabilities to do a wide variety of tasks, such as stealing secret or personal information, making a profit, or just enjoying.

In the past, most prevalent attacks are ones to applications in binary code on the client, as represented by *buffer overflow attacks*. This kind of attacks, however, has been subsided. It is possibly because most of them can be prevented by *NX bit* and *ASLR(Address Space Layout Randomization)*.

Instead of them, the most serious attacks in recent years are *script injection attacks* to web servers, such as *cross-site scripting* (*XSS*), *SQL injection* or *directory traversal*. According to National Vulnerability Database[1], vulnerabilities to script injection attacks have been increased sharply in recent years.(**Fig. 1**).

DTP (Dynamic Taint Propagation) is proposed to prevent these attacks. The mechanism of DTP is to tag data from untrusted sources as *tainted*, dynamically propagate tainted information with program execution and check whether the tainted data is an attack or not.

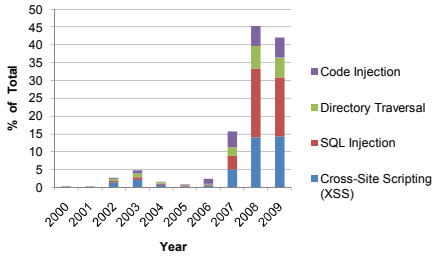

Fig. 1 Increase of script injection attack

The original inspiration of DTPs was given by the taint mode of Perl[2]. Since then, this kind of techniques have been supported by various programming language systems, such as PHP[3)~5)], Ruby, Java[6),7)], C[8),9)] and its descendants. These language-level supports are referred to as *language DTP*s. On the other hand, Suh et al. first applied Perl taint mode to a processor in order to detect injection attacks to binary code, and named it DIFT(Dynamic Information Flow Tracking)[10]. 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 they could provide a accuracy class which is just the same as language DTPs[11] .

Though DTPs are considered to have potential to root out script injection attacks, current systems still suffer from trade-off between false positives and negatives. We take Base64 as an example in order to explain the trade-off problem in section 2. Existing DTPs don't propagate tainted information through Base64. This is because Base64 converts source strings by using table reference. If we regard table reference as safe, it produces security hole. On the contrary, if we regard table reference as unsafe, it results in mass of false positives. Conventional DTPs select the former and they don't propagate tainted information through Base64.

We proposed a technique named **SWIFT**[12],

† Graduate School of Information Science and Technology, The University of Tokyo
†† Graduate School of Engineering, Nagoya University

which provides a solution to deal with the trade-off problem. We introduced a completely different approach from conventional systems. As is referred to in more detail in Section 3, SWIFT observes the memory access of the target program, detects string access from address trace, distinguish string operations from common memory accesses, and propagate tainted information through string operations from read string to write string. We showed the result that SWIFT provided a better accuracy on detecting script injection attacks.

We proposed SWIFT on processors as platform DTP[12], because the main advantage of platform DTPs is the comprehensiveness. However, it is hard for SWIFT to be commonly used because it is difficult to design new processors.

In this paper, We implemented SWIFT to PHP as an example of language DTP. We selected PHP as an implementation target because PHP is widely used in the world as script language designed for server-side web applications. By implementing SWIFT to PHP, coverage is limited for PHP, but the highly accurate DTP can be implemented in the range of taint-support PHP. And it is easy for SWIFT to be broadly used.

The rest of the paper is organized as follows. Section 2 reviews background knowledge for script injection attacks and DTPs. Section 3 firstly describes two types of string operations in order to explain why existing DTPs fall into trade-off between false positives and negatives. In the rest of this section, we describe SWIFT in detail, that is free from the trade-off problem. Section 4 explains how to implement SWIFT to PHP. Section 5 summarizes the evaluation results of the accuracy and the performance overhead. Section 6 presents the related work. Section 7 states the conclusion and future work.

2. Script Injection Attacks and Dynamic Taint Propagation

2.1 SQL injection

From cross-site scripting to SQL injection, attackers have various techniques to attack web applications. This subsection takes SQL injection as an example to explain how script injection attacks occur.

SQL injection is a most common attacks. It allows an attacker to access sensitive information from a Web server's database. **Fig. 14** shows an example of SQL injection.

A web page shows the price of a product asking the user the name of it through a text box. **Fig. 2(a)** shows a PHP statement in the page. The string the user entered in the text box has been stored in the variable `$name`. Concatenating `$name` and the constant strings, the statement produces the SQL

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

(a) PHP statement

```
$name:
  ruby

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

(b) Non-attack string and produced 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 produced command

Fig. 2 Example of SQL injection

command `$cmd` to send to the SQL server.

In a usual case, the user entered just `ruby` for `$name`, `$cmd` in **Fig. 2(b)** is produced. In this and the next figures, the substrings corresponding to `$name` are underlined. The database will return the price of `ruby`.

If an attacker injects the string into `$name` as in **Fig. 2(c)**, `$cmd` in the same figure is produced. Then, the database will be updated against the programmer's intention.

As seen in this example, a script injection attack is performed by making the victim server interpret the string including attack code written in script language. As for binary injection attacks, even if an attack binary is successfully injected, execution of injected binary can be easily prohibited, e.g., by NX bit. As for script injection attacks, however, interpretation of injected scripts itself cannot be prohibited, because it is the main benefit in using script languages. This is the main difficulty of script injection attack detections.

2.2 Detection of Script Injection Attacks

DTPs are promising techniques to detect script injection attacks. The idea behind DTP is to tag data from untrusted source as tainted. The tag is used to tainted data from untrusted source, for example, data from network I/O, user input, or read from any untrusted devices. The tags are propagated through program execution. If tainted data is used in unsafe ways, such as a system call or a SQL command, attacks will be detected.

In the example of SQL injection in section 2.1, tainted substring is underlined. Because SQL commands, such as *UPDATE* or *SET*, or field and table names, such as *price* or *prod* are tainted, SQL in-

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

(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

Fig. 3 Example of base64 vulnerability

jection can be detected.

In the next subsection, we explain command parsing, which SWIFT is supposed to be used with.

2.3 Command Parsing

Su et al. show that SQL injection can always be perfectly detected as long as the SQL syntax is known and the substrings are correctly detected *trusted* or *untrusted*[13].

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

This command parsing can also be applied to any commands raised from web applications other than SQL such as system calls. In general, data from untrusted source should not specify the names of the system resources, but may specify their contents. The names of the 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 on its own judgment. DTPs should always leave the substring corresponds to `$name` tainted even in non-attack cases. In the example of section 2.1, even if `ruby` is left tainted, the parser can distinguish it from attacks. Academic researcher have had appropriate command parsing to prevent many kinds of script injection attacks[14][9].

In the next subsection, We take Base64 as an example to explain a problem of existing DTPs.

2.4 Problem of Existing DTPs

Some web applications use Base64 to obfuscate sensitive input. In Cubecart3.0.3, we could find the code in **Fig. 3(a)**

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

For example, if an attacker creates and inputs a specially crafted URL in **Fig. 3(b)**, `$redir` in **Fig. 3(c)** is generated after *base64_decode* function. And when the code is executed, cross-site scripting occurs.

Existing DTPs don't propagate tainted information through Base64, so they can't detect the cross-site scripting mentioned above. In the rest of this section, we will explain why existing DTPs don't propagate tainted information through Base64.

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 by using a conversion table

Base64 decoding procedure is the reverse. The point is that Base64 uses table reference as conversion. In general, table reference is like this:

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

We can regard table reference as safe in usual cases, but if it is used as conversion, it is unsafe. When thinking from the point of taint propagation, table reference falls into a trade-off between false positives and negatives. If we regard table reference as safe, tainted information isn't propagated from $istr to $ostr, and it produces security hole. On the contrary, if we regard table reference as unsafe, tainted information is propagated from $istr to $ostr, and it results in mass of false positives. Most existing DTPs select the former, so they don't propagate tainted information through Base64.

3. SWIFT

This section explains SWIFT that we have proposed[12]. SWIFT provides a higher accuracy on detecting script injection attacks than conventional DTPs. First, we describe two types of string operations: "loop-in-select" and "select-in-loop" in section 3.1. Second, we present the algorithm of SWIFT in detail in section 3.2.

3.1 Loop-in-Select and Select-in-Loop

Fig. 4 shows that string operations using table reference can be classified into two types: safe and unsafe. The string operation in **Fig. 4(a)** is safe, because the string that the user chooses is necessarily under control of the programmer. It is practically impossible for attackers to attack through this operation. On the other hand, the string operation in **Fig. 4(b)** is unsafe, because the user can control the output and get arbitrary string. It is possible to attack through this operation. This string operation includes string copy and all kinds of string conversions such as case conversions or coding conver-

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

(a) Sample code of safe string operation

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

(b) Sample code of unsafe string operation

Fig. 4 Two types of string operations

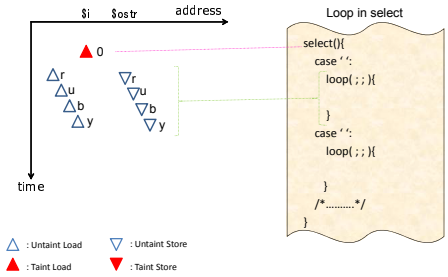

Fig. 5 "Loop-in-Select" structure - safe

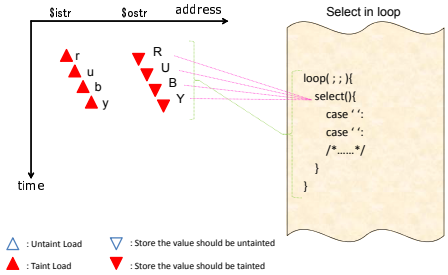

Fig. 6 "Select-in-Loop" structure - unsafe

sions.

The string operations in Fig. 4(a) and Fig. 4(b) are almost the same except that the table reference of the latter unsafe string operation is used in the for statement. We can distinguish these two types by focusing on address trace. **Fig. 5** and **Fig. 6** show address traces of these string operations. In these figure, the x-axis indicates the address, and the y-axis indicates the time. There are four types of triangles. Upward blue triangles and red triangles indicate load instructions to untaint data and taint data, respectively. Downward blue triangles and red triangles indicate store instructions whose store value should be untainted and tainted, respectively. The load/store instructions that do not relate to DTP are not drawn in these figures.

Fig. 5 corresponds to the sample code of the safe string operation shown in Fig. 4(a). The first tainted load indicates the load to input variable `$i`. In this case, the value of `$i` is `'0'`, then the constant string `"ruby"` is copied to the output variable `$ostr`. Fig. 6 corresponds to the sample code of unsafe string operation shown in Fig. 4(b). The tainted input string `"ruby"` is converted to `"RUBY"`.

In the 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 while tainted in the unsafe string operation. If we regard table reference as "select" and interleaving read/write as "loop", we can see that the safe string operation is **loop-in-select** structure in Fig. 5. And, we can also see that unsafe string operation is **select-in-loop** structure in Fig. 6. Existing DTPs don't pay attention to non-local structures such as loop-in-select and select-in-loop, so they fall into table reference trade-off.

3.2 Loop-in-Select Structure Detection

3.2.1 Abstract of Algorithm

We proposed SWIFT[12], which is a proper method to distinguish loop-in-select and select-in-loop structures. Unlike existing DTPs, SWIFT doesn't track information flow instruction by instruction. SWIFT only observes address trace of executed load/store instructions and detects select-in-loop string operations.

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

3.2.2 Streams and Interleaving Pair

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. **Fig. 7** 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 in turn. The read stream is divided into plural read **substreams** by occurrences of the stream writes, and vice versa. Each of the read/write substreams contains one or more stream reads/writes. And, a read/write access in the read/write stream of an interleaving pair is referred to as an **interleaving-stream read/write**.

3.2.3 Tables

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

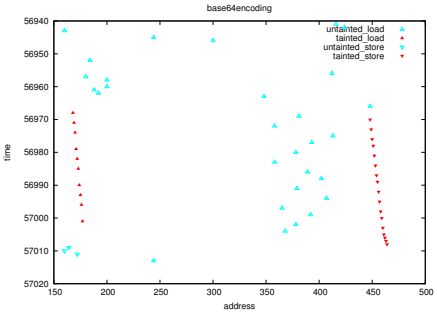

Fig. 7 Address trace of *base64_encode*

The entry of 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*.

3.2.4 Stream Read/Write Detection

On a read access to *addr*, *next* of all the entries of 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 to represent 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. And, the same holds true for the write table and write accesses.

3.2.5 Interleaving Stream Read/Write Detection

When a stream write is detected, the *switched* flags of all the entries of the **read** (not write) table are set. After that, 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 the *next* and *n_substrm* is greater than a threshold, the read access is detected as an interleaving stream read. Likewise, the same holds true for the write table and write accesses.

3.2.6 Propagation and Backtracking

Every time a stream read is detected, the taintedness of the read is stored in the *taintedness*. 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 mainly used to locate the start address of the stream.

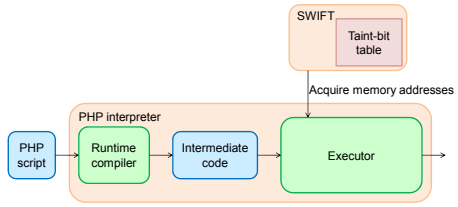

Fig. 8 General view of implementation

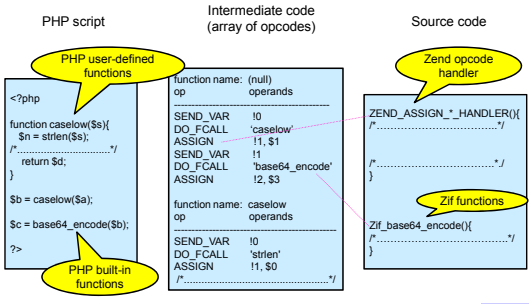

Fig. 9 Relationship among script, intermediate code and source code

4. Implementation of SWIFT to PHP

This section explains how to implement SWIFT to PHP. Since SWIFT only focuses on address traces of a program execution, it can be implemented both on script interpreters and on processors.

PHP is widely used in the world as script language designed for server-side web applications. By implementing SWIFT to PHP, coverage is limited for PHP, but a highly accurate DTP can be implemented in the range of taint-support PHP. And it is easy for SWIFT to be popularized.

We first give an overview of implementation in section 4.1. After that, we briefly summarize ease of taint propagation in interpreters and native functions from the point view of platform DTPs and language DTPs in section 4.2. In section 4.3, we explain PHP interpreter. Finally, we explain how to acquire memory addresses on the source code of the interpreter in detail in section 4.4.

4.1 Overview

Fig. 8 shows the general view of our implementation. A script of PHP is compiled to the intermediate code by a runtime compiler and is executed by an executor. We have already had SWIFT engine, which has hash table of taint-bit whose access keys are memory addresses. What we have to do is to get memory addresses from a source code of the executor by inserting hook functions.

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

Fig. 10 Zval structure

4.2 Ease of Taint Propagation: interpreters and native functions

In platform DTPs, it is difficult to propagate tainted information through an interpreter because there is mass of interpreting noise, while it is easy to do through native functions because there is no influence of interpreting noise. Interpreting noise is instructions executed only for interpreting scripts, which provide no help in information flow tracking. Tens of native instructions are needed just in order to interpret a single instruction of the intermediate language of scripts. And these instructions are not directly related to information flow tracking, so for the script injection attack detection, it behaves as noise.

In language DTPs, however, ease of taint propagation mentioned above is reversed. It is easy to propagate tainted information through an interpreter because we can utilize interpreter's information, while it is difficult to do through native functions because we can't utilize interpreter's information. In this research, we acquire memory addresses from each native functions reading source code.

4.3 PHP Interpreter

4.3.1 Relationship among Script, Opcode and Source code

We use **Fig. 9** in order to describe relationship among scripts, intermediate codes and source codes of PHP interpreter. The left script in Fig. 9 is a sample script.

The intermediate code is an ordered array (an *op array*) of instructions (known as *opcodes*)[15], such as *DO_FCALL* and *ASSIGN*. We call source codes of each opcodes **zend opcode handlers**.

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

The sample script also uses *caselow*, which is a *PHP user-defined function*. In an intermediate code, PHP user-defined functions are 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*. **Fig. 10** and **Fig. 11** show zvals C structure and its complementary data container.

To access the data in the various types, you can use the macros, which take zvals as their arguments. For instance, if you want to extract the string buffer for *zval*, *zval** or *zval***, you would use *Z_STRVAL*, *Z_STRVAL_P* or *Z_STRVAL_PP*.

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

Fig. 11 Zval's data container

Table 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

4.3.3 Memory Management

PHP interpreter uses its own internal memory-management wrapper functions. **Table 1** shows memory-management wrapper functions.

4.4 Acquisition of Memory Addresses

We explain how to acquire memory addresses. Because we can utilize information of the interpreter's source code, there is no influence of interpreting noise, namely we can acquire only memory addresses of strings. We acquire memory addresses from zend opcode handlers and zif functions.

4.4.1 Zend Opcode Handler

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
- PRE_INC, PRE_INC_OBJ

We focus on zend opcode handlers corresponding to these opcodes. In zend opcode handlers, string access is done by using the macros mentioned in section 4.3.2, so we can recognize string access. However, we don't get memory addresses from all macros, because the macros only return the pointer to the string. We get memory addresses only when a memory area where string is stored moves to another memory area. There are two actual cases.

One case is that memory-management wrapper functions take the macros as their arguments. The function to which we should pay attention is the following: *estrndup*, *erealloc*, *memcpy*. *Memcpy* is *C built-in library function*. *Estrndup* and *erealloc* use *memcpy* internally. Address trace of *memcpy* is interleaving read and write. For example, we can find source code as below:

```
PHPAPI unsigned char *php_base64_encode
(const unsigned char *str, int length, int *ret_length)
{
  const unsigned char *current = str;
  unsigned char *p;
  unsigned char *result;
  /* ... */
  while (length <2) {
    swift_load((char*)current,1);
    swift_store((char*)p,1);
    *p++ = base64_table[current[0] >>2];
    swift_load((char*)&current[1],1);
    swift_store((char*)p,1);
    *p++ = base64_table[((current[0] & 0x03)
     <<4) + (current[1]>>4)];
  /* ... */
}
```

Fig. 12 Acquisition of memory addresses from zif function

```
memcpy(Z_STRVAL_P(result), Z_STRVAL_P(op1),
Z_STRLEN_P(op1));
```

In this case we should get addresses of *Z_STRVAL_P(result)* and *Z_STRVAL_P(op1)*. *Z_STRVAL_P(op1)* corresponds to read string, while *Z_STRVAL_P(result)* corresponds to write string.

Another case is that the macros access an element of the string by using subscript. For example, we can find source code as below:

```
Z_STRVAL_P(T->str_offset.str)[T->str_offset.offset]
= Z_STRVAL(tmp)[0];
```

In this case we should get the addresses of right and left operands as read access and write access.

4.4.2 Zif Functions

There are about a hundred zif functions from which we have to get memory addresses. For example, *urlencode*, *base64_encode*, *ereg_replace*. Because only pointers to char are passed to zif functions, we must read the source code and get memory addresses mechanically . In source code of *base64_encode*, we get memory addresses like **Fig. 12**

Reading the source code, we find that the variable `current` corresponds to read string and the variable `p` corresponds to write string. We use *swift_load* and *swift_store* as hook functions to get memory addresses. The first argument of these functions is memory addresses of load/store access and the second arguments is bytesize of load/store access.

5. Evaluation

5.1 Evaluation Method

5.1.1 Environment

We implemented SWIFT to PHP-5.3.1. As for PHP-taint, we used PHP-taint 20080622 package[3]. PHP-taint is first implementation of taint support for PHP released November 2007. **Table 2** shows the evaluation environment we set.

5.1.2 Methodology

To evaluate the taint propagation accuracies, we checked which substring of the output string is tainted when the target programs call sensitive functions.

Table 2 Evaluation environment

PHP-SWIFT	modified PHP-5.3.1
PHP-taint	PHP-taint 20080622 package
OS	Ubuntu 9.04
Web Server	Apache 2.2.14
SQL Server	Mysql 5.1.37

To evaluate the performance overhead, we inserted *microtime* function, which is a PHP built-in function, into first and last of the scripts and measured execution time of interpreting scripts.

5.2 Taint Propagation Accuracy

5.2.1 String Operations

Table 3 summarizes the result of basic string operations. The string operations include string copies, case and code conversions, which are commonly used in web applications. (2) to (7) are PHP built-in functions, thus they are written in C.

(1)concatenation, (2)*substr()*, and (3)*ereg_replace()* execute string copies in the ends of operations, and all the models can propagate taint correctly. (4)*ereg()* is regular expression match, and all the model untaint the scalar result.

(5)*strtoupper/strtolower()* are case conversions.

(6)*urlencode/urldecode()* and (7)*base64_encode/ base64_decode()* do encode and decode operations. As a result, PHP-taint untaints the outputs of all these functions.

(8)untaint table and (9)taint table retrieve values from tables with taint keys.(8)untaint and (9)taint table have been stored untaint and taint values, respectively. Since PHP-taint regards the values from tables as safe, it results in false negative in (9)taint table. On the other hand, PHP-SWIFT can track the flow between the input and the output values through a table.

(10) is a uppercase conversions code, shown in Fig. 4(b). Though the function is the same as (5)*strtoupper()*, it is written in PHP. (10) is written with a switch statement construction. PHP-SWIFT produce no false positives or negatives, because PHP-SWIFT can correctly propagate tainted information for all the operations. So, even if programmers use operations such as these to be the input arguments of applications, PHP-SWIFT could also provide high precision.

5.2.2 Real-World Web Applications

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. To choose the web applications, we selected applications whose specific exploit codes can be found on the web. These applications use some input variables as an argument without validation or even any string operations to them. We made Script injection

Table 3 Results of string operation

	Operation	PHP-SWIFT FN	PHP-SWIFT FP	PHP-taint FN	PHP-taint 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

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

(a) Attacke code

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

(b) Path argument

Fig. 13 Qwikiwiki vulnerability

attacks such as cross-site scripting (XSS), SQL injection and directory traversal according to the exploit code. As summarized in **Table 4**, PHP-SWIFT caused no false positives or negatives. But PHP-taint 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 make the attack code to be base64 encoded, and use these base64 encoded code to create a specially crafted URL.

The vulnerabilities of other web applications are elementary.

In the rest of this section, we will 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.

For example, the attack code is configured as **Fig. 13(a)**. Then, it raised *open()* system call with the string in **Fig. 13(b)** as its path argument. It will open *_config.php* that an attacker is not allowed to access. PHP-SWIFT successfully tainted the underlined substring.

Qwikiwiki first stores the parameters such as *page* into a hash table, then it uses them from the table. Therefore, the same situation as (9) taint table in the previous subsection occurs.

PHP-Nuke 7.1

PHP-Nuke 7.1 has a SQL injection vulnerability in ”modules.php”. When we check the source code of ”modules.php”, we could find the code in **Fig. 14(a)**.

```
$nukeuser =
  base64_decode($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

Fig. 14 PHP-Nuke vulnerability

Table 4 Results of web applications

Program	Attack	PHP-SWIFT FN	PHP-SWIFT FP	PHP-taint FN	PHP-taint 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			√	
CubuCart 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 Performance overheads

Program	Attack	Overhead
phpSysInfo 2.3	Cross-site scripting	6%
Qwikiwiki 1.4.1	Directory traversal	95%
phpBB 2.0.8	Cross-site scripting	72%
PHP-Nuke 7.5	SQL injection	35%
CubuCart 3.0.3	Cross-site scripting	84%
PHP-Nuke 7.1	Cross-site scripting	53%
PHP-Nuke 7.1	SQL injection	36%

And we can see the base64 decoded global variable `$nukeuser`, the application did nothing to validate the variable. So it means the variable `$nukeuser` can contain user supplied data with out sanitization and lead to sql injection.

For example, if an attacker input a specially crafted URL in **Fig. 14(b)**, `$nukeuser` in **Fig. 14(c)** is generated after *base64_decode* function. Concatenating the substring of `$nukeuser` and the constant strings a programmer provided, an attack sql query is produced.

In the issue, we could bypass the user-level authentication by this exploit.

5.3 Performance

Table 5 shows the performance overheads for

web applications against an unmodified version of PHP. The overhead is between 6% to 95%, with an average of 55%. This overhead is bigger than prior result reported by PHP-taint[3]. Optimization of implementation is our future work.

6. Related Work

6.1 Existing DTPs

This subsection summarizes existing DTPs from the viewpoint of **non-propagation policy**. Non-propagation policy is a policy when NOT to propagate tainted information. It is as important as how to propagate tainted information, because a perfect DTP, which can perfectly track all kind of information flow(data flow, address flow, and control flow), would mark all the output as tainted and such DTP is useless.

As far as we know, all the current DTPs define non-propagation policyon heuristics, such as about sanitization or table reference.

Perl taint mode

As described before, the notion of DTP was first introduced by taint mode of Perl[2].

Perl's Taint Mode is a collection of specific restrictions to help programmers to write safer scripts. While in this mode, Perl takes special precautions called taint checks to prevent security traps which originate from external data. It make the external data be tainted, and tainted data is prevented from being used directly or indirectly in any conditions which contains potential hazard, such as invoke a sub-shell, modify files or directories, send sensitive data over network. Since it is the first implementation of DTP, the non-propagation rules are not so sophisticated.

The only way to clean taint bits of variables is to do a regular expression match. Subpatterns from regular expression matches and values used as keys in a hash table are untainted. And it does not track control flows.

DTP for Java string

Haldar et al. implemented a modified *java.lang.String* for DTP[6]. It specifies sources and sinks for the J2EE library. And it associated a taint flag with every string to track the taintedness of strings. It taints strings and propagates tainted information from sources to sinks. Strings are untainted when passed through regular expression matches, or been tested for the presence of a particular character.

It trust the programmer to have performed a meaningful check that accounts for all cases that might be exploitable in an attack. But the most commonly vulnerabilities are caused by programmer's faulty input validations or faulty coding. So it could not provide a precise detection for script injection attacks.

Raksha

Raksha is one of the latest descendants of platform DTP. Raksha architecture provides a flexibility DTP proposal for taint propagation[11]. And the most significant attribute of Raksha is it points out platform DTP systems can also detect script injection attacks.

It supports multi-bit taints and extends taint propagation registers which can follow four different policies. Processor provides a comprehensive platform, so it can work with arbitrary binary and independency of code languages. It could deal with a wide range of web application attacks and work with any programs which written in multiple code languages.

The main non-propagation policy of Raksha is to untaint when bounds check is detected, more precisely, to untaint a register when it is compared. And Raksha does not track address flows by default, in order not to propagate tainted information through hash tables or arrays. Although Raksha can be configured to track address flows, it results in mass of false positives[11].

As just the same as the other DTPs, it does not track control flows.

Source transformation for C

Xu et al. applies a taint analysis to a source-to-source transformation of C programs targeting not only for binary but also for script injection attacks[8),9)]. It can transform script language interpreters written in C into ones resistant to injection attacks.

Though the approach itself is very comprehensive, the non-propagation policy is quite poor. For example, it propagate tainted information through hash tables.

String conversions, such as case conversions or coding conversions are very commonly used in web applications. When the source string is tainted, the converted string should also be tainted. Such code conversions, however, sometimes utilize hash tables.

It is still difficult even for the static analyzer to distinguish string conversions using hash tables because syntactic structures of the target programs are complicated. Therefore, they decided to propagate tainted information even through hash tables, which leads to a mass of false positives.

6.2 Blackbox Detection

Prithvi et al. develop an approach and a tool to identify server-side input validation problems, especially parameter tampering vulnerabilities, using a black-box analysis of the server[16]. The tool is called NoTamper. First of all, NoTamper analyzes each form on the web page and constructs logical formulas representing the constraint-checking function. Then, it submits hostile and benign inputs to

the server, and compare each hostile response to benign responses. Finally, a human tester verify hostile inputs as parameter tampering vulnerabilities.

Though this approach itself is comprehensive, it cannot offer guarantees of completeness because of the inherent limitations of black-box analysis. In addition, the approach can only detect parameter tampering vulnerabilities, while DTPs, including SWIFT, can detect various kinds of server-side input validation attacks, which can execute arbitrary malicious codes.

As a result of their evaluation, they were able to confirm only 9 exploits out of 50 tampering opportunities. 43 cases are false positives. The false positive rate is very high. Moreover, examining 50 opportunities takes time.

7. Conclusion

In this paper, we implemented SWIFT to PHP. SWIFT is a completely different approach from conventional DTPs. In order to detect script injection attacks precisely, SWIFT observes memory accesses of a target programs, detects select-in-loop string operations and propagate tainted information through them. Since SWIFT only uses address traces of a program, it can be implemented both on script language interpreters and on processors.

We implemented SWIFT to PHP, compared the accuracy with taint-support PHP and evaluated the performance overhead. PHP-SWIFT can correctly propagate tainted information for typical string operations and real-world web applications with known vulnerabilities, while PHP-taint don't. And the average performance overhead is 55%.

In this research, we did naive implementation of SWIFT to PHP. Thus, we are interested in optimization of implementation so as to reduce the performance overhead. This is our future work.

Another plan for the future work is to do further evaluation using real-world web applications without known vulnerabilities. We want to verify no false positives are produced.

We are planning to distribute PHP-SWIFT in the near future.

Acknowledgments

This research was partially supported by Core Research for Evolutional Science and Technology (CREST) of the Japan Science and Technology Agency (JST), and by Grant-in-Aid for Scientific Research (B) No.22300014 from Ministry of Education, Culture, Sports, Science and Technology Japan.

References

1) NIST: National Vulnerability Database. http://web.nvd.nist.gov/view/vuln/statistics.
2) Allen, J.: Perl Version 5.8.8 Documentation - Perlsec (2006). http://perldoc.perl.org/perlsec.pdf.
3) Venema, W.: Taint support for PHP (2008). http://wiki.php.net/rfc/taint.
4) Pietraszek, T. and Berghe, C.: Defending against Injection Attacks through Context-Sensitive String Evaluation, *8th Int'l Symp. on Recent Advances in Intrusion Detection*, pp. 124–145 (2005).
5) Nguyen-Tuong, A., Guarnieri, S., Greene, D., Shirley, J. and Evans, D.: Automatically Hardening Web Applications using Precise Tainting, *20th Int'l Information Security Conf.* (2005).
6) Haldar, V., Chandra, D. and Franz, M.: Dynamic Taint Propagation for Java, *21st Annual Computer Security Applications Conf.* (2005).
7) Livshits, B., Martin, M. and Lam, M. S.: SecuriFly: Runtime Protection and Recovery from Web Application Vulnerabilities, *Tech. Rep., Stanford Univ.* (2006).
8) Xu, W., Bhatkar, S. and Sekar, R.: Practical Dynamic Taint Analysis for Countering Input Validation Attacks on Web Applications, *Tech. Rep. SECLAB-05-04* (2005).
9) Xu, W., Bhatkar, S. and Sekar, R.: Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks, *15th USENIX Security Conf.*, pp. 121–136 (2006).
10) Suh, G. E., Lee, J. W., Zhang, D. and Devadas, S.: Secure Program Execution via Dynamic Information Flow Tracking, *11th Int'l Conf. on Architectural Support for Programming Languages and Operating System* (2004).
11) Dalton, M., Kannan, H. and Kozyrakis, C.: Raksha: A Flexible Information Flow Architecture for Software Securit, *34th Int'l Symp. on Computer Architecture*, pp. 482–493 (2007).
12) Li, K., Shioya, R., Goshima, M. and Sakai, S.: String-wise information flow tracking against script injection attacks, *IEEE Int'l Symp. on Pacific Rim Dependable Computing (PRDC 2009)*, pp. 169–176 (2009).
13) Su, Z. and Wassermann, G.: The Essence of Command Injection Attacks in Web Applications, *33rd Symp. on Principles of Programming Languages* (2006).
14) Dalton, M.: *THE DESIGN AND IMPLEMENTATION OF DYNAMIC INFORMATION FLOW TRACKING SYSTEMS FOR SOFTWARE SECURITY*, PhD Thesis, Stanford University (2009).
15) Schlossnagle, G.: *Advanced PHP Programming*, DEVELOPER'S LIBRARY (2003).
16) Bisht, P., Hinrichs, T., Skrupsky, N., Bobrowicz, R. and Venkatakrishanan, V.: NoTamper: Automatic Blackbox Detection of Parameter Tampering Opportunities in Web Application, *CCS '10 Proceedings of the 17th ACM conference on Computer and communications security*, pp. 607–618 (2010).