Vol.2010-OS-115 No.4
2010/8/3

文字列ごとの情報フロー追跡手法の PHP への実装と評価

都井紘[†1]　塩谷亮太[†1]
五島正裕[†1]　坂井修一[†1]

近年，クロス・サイト・スクリプティングや SQL インジェクションといった Web アプリケーションの脆弱性を突いたインジェクション・アタックによる被害が深刻化している．インジェクション・アタックを検出する方法として DTP(Dynamic Taint Propagation) が研究されている．DTP では，外部からの入力にテイント（汚染）をつけ，演算の入力から出力に伝播させ，最後にテイントのついたデータが ’危険 ’な使われ方をしないかチェックする．従来の DTP では，命令間のデータの依存関係に基づいてテイント情報を伝播していたため，検出漏れと誤検出のトレードオフに陥り，伝播精度が十分ではなかった．

そこで李らは，load/store 命令のメモリアクセスから文字列操作を識別し，文字列から文字列へテイント情報を伝播させる SWIFT を提案している．SWIFT はこのようにローカルでない追跡を行うため，その伝播精度は従来の DTP より高いことが示されている．しかし，SWIFT はハードウェア上に実装するため普及のハードルが高い．

本稿では，SWIFT を Web 用スクリプト言語として現在最も多く用いられている PHP 上に実装し，典型的な文字列操作を行うプログラム，および脆弱性の確認されている Web アプリケーションにおいて正しくテイント情報を伝播することを確認した．

Implementation and Evaluation of String-Wise Information Flow Tracking to PHP

HIROSHI TOI,[†1] RYOTA SHIOYA,[†1] MASAHIRO GOSHIMA[†1]
and SHUICHI SAKAI[†1]

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) and DIFT (Dynamic Information Flow Tracking) have been established as powerful techniques to detect script injection attacks. However, current DTP/DIFT systems still suffer from trade-off between false positives and negatives, because these systems propagate taint from source to destination operands.

So Li et al. propose String-Wise Information Flow Tracking, SWIFT. SWIFT traces memory access of a program execution, detects string access and distinguishes string operations from other memory access, and propagates taint information under string operations. This makes SWIFT provide a better accuracy on detection of script injection attacks than current DTP/DIFT systems. Since SWIFT concentrates on address traces of a target program, it can be implemented both on interpreters of script languages and on 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, SWIFT on PHP shows a high precision in our experiments.

1. Introduction

Increase in web applications leads to increase in security incidents. The internet has become the primary conduit for attack activities, and vulnerabilities of web applications are getting more attentions. The attackers exploit diversified security vulnerabilities to accomplish a wide variety of malicious tasks, such as steal of secret or personal information, making a profit, or just for fun.

In the past, most predominant 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*.

Instead of them, the most serious attacks in recent years are *script injection attacks* to web servers, such as *directory traversal*, *Remote File Inclusion* (*RFI*), *Cross-site scripting* (*XSS*), or *SQL injection*. CVE reported vulnerabilities to script injection attacks have been increased sharply in recent years[3]. This is probably due to proliferation of low-grade applications written by inexperienced developers, and ease of exploitation of the vulnerabilities.

DTP (Dynamic Taint Propagation) and **DIFT** (Dynamic Information low Tracking) are proposed to prevent these attacks. The idea behind DTP and DIFT is to tag data from untrusted sources as *tainted*, propagate taint information and check tainted data.

Though DTP/DIFT are considered to have potential to root out script injection attacks, current systems still suffer from tradeoff between false positives and negatives.

So li et al. proposed a technique named **String-Wise Information Flow Tracking**, **SWIFT**. They introduce a completely different approach from conventional systems. SWIFT observes the memory access of the target program, detects string access from address trace, distinguish string

†1 東京大学大学院　情報理工学系研究科
Graduate School of Information Science and Technology, The University of Tokyo

operations from common memory access, and propagates taint through string operations from load string to store string. Since SWIFT only concentrates on address traces of a target program, it can be implemented both on interpreters of script languages and on processors. Li et al. proposed SWIFT on processors, but it is hard for SWIFT to be familiarized.

In this paper, We implemented SWIFT to PHP and executed typical string operations. PHP is widely used in the world as scripting language that is designed for server-side web development. By implementing SWIFT to PHP, coverage is limited for PHP, but the highly accurate DTP can be made in the range of taint-support PHP. And it is easy for SWIFT to be familiarized.

The rest of the paper is organized as follows. Section 2 reviews background knowledge for script injection attacks. Section 3, 4 and 5 summarizes current DTP/DIFTs and taint propagation algorithms. Then, section 6 explains SWIFT in detail. Section 7 and 8 explains how to SWIFT to PHP and evaluation. Section 9 states the conclusion.

2. Script injection attacks

From Cross-Site Scripting to SQL injection, hackers have various techniques at their disposal to attack Web applications. This section takes SQL injection as an example to explain how script injection attacks occur.

SQL injection is a most popular attacks. It allows an attacker to access sensitive information from a Web server's database. Next we explain the mechanism of SQL injection by using the following example. A web page shows the price of a product asking the user the name of it through a text box. The below code shows a PHP statement in the page

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

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

In a usual case, for example, the user entered just **ruby** for **$name**, the following code is produced as:

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

Then the database will return the price of **ruby** to client. If an attacker injects the string into **$name** by using the code like:

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

Then following SQL query will be produced.

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

As a result, the database will be updated against the programmer's intention by the attacker.

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 attack, 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 attack, however, interpretation of injected scripts itself cannot be prohibited, because it is the main benefit in using scripts. This is the main difficulty of script injection attack detections.

3. DTP and DIFT

The original inspiration of this area was given by the *taint mode* of Perl. The main purpose was to prevent script injection attacks especially to web applications[1].

Since then, this kind of techniques have been supported by various programming language systems, such as PHP, Ruby, Java, C and its decendants[5–9,12]. These language-level supports are often referred to as *DTP* (Dynamic Taint Propagation).

On the other hand, Suh et al. applied Perl taint mode to a processor in order to detect injection attacks to binary code, and named it *DIFT* (Dynamic Information Flow Tracking)[11]. Nowadays, the name of DIFT is often used to refer to all such techniques on processors[2].

Although the purpose of DIFT was to detect binary injection attacks, DIFT can also be used to detect script injection attacks[4]. In this paper, we discuss DTPs and DIFTs in a unified viewpoint.

3.1 Advantage and disadvantage of DIFT

The key advantage of DIFT in script injection attack detection is the comprehensiveness. DTP systems are implemented on language-level, so they are language-specific. For example, Perl taint mode cannot be used for other languages such as PHP. But DIFT systems are not language-specific. Therefore, DIFT provides a more comprehensive platform than DTP. And obviously DIFT is preferable script injection attack detection than DTP if its accuracy is as high as DTP.

However, until now, for the present DIFT systems, the detection accuracy is considered as a

lower one than that of DTPs. Because DIFT could not utilize information of script or high-level languages as a support on detection. In addition, not only the mass of instructions which are executed during interpreting the script provides no helps in information flow tracking, but also it behave as noise.

The two facts are supposed as the main reasons which degrade the accuracy to detect attacks. But, evaluation results show that DIFT could provide a accuracy class which is just the same as DTP[4]. The reason is that the propagation algorithms of DTP/DIFT make more influence on their accuracies than above factors and by using proper propagation algorithms DIFT could get a good enough detection accuracy. In next section, we will explain this point of view by discussing information flows in program execution and taint propagation algorithms in detail.

4. Current problem

Some Web applications use Base64 to obfuscate sensitive input. In Cubecart3.0.3, we could find the code as below:

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

After this base64_decode(), $redir is not sanitized, and this can lead to a remote Cross-site scripting attack. For example, if a user creates and inputs a specially crafted URL like:

```
http://[victim]cc3/index.php?act=login&redir=L3NpdGUvZ
GVtby9jYzMvaW5kZXgucGhwP2FjdD12aWV3RG9jJmFtcDtkb2NJZD0x
```

And the base64encoded part of variable redir is

```
L3NpdGUvZGVtby9jYzMvaW5kZXgucGhwP2FjdD12aWV3RG9jJmFtcDtkb2NJZD0x
```

After base64_decode function, it will generate a code like this:

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

And when the code is executed, it will cause a remote Cross-site scripting.

Existing DTP/DIFTs don't propagate taint through Base64, so they can't detect the Cross-site scripting mentioned above. In the rest of this section, we will explain why existing DTP/DIFTs don't propagate taint 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. If we regard table reference as safe, taint isn't propagated from $istr to $ostr, and it produces security hole. On the contrary, if we regard table reference as unsafe, taint is propagated from $istr to $ostr, and it results in mass of false positives. Most existing DTP/DIFTs select the former, so they don't propagate taint through Base64.

5. Taint propagation algorithms

This section summarizes the algorithms of taint propagation. Firstly we summarize types of information flow and their taint propagations. Next, subsection 5.2 describes non-propagation policy, which is one of the most important factors of propagation algorithms.

5.1 Types of information flow and taint propagation

Information flow can be divided into data, address, and control flows.

Data flow is associated with direct data dependence. In the following sample code, there are direct data dependences from the right to the left hand objects.

```
$o = $i;            // $o depends on $i
$o = $i + $j;       // $o depends on $i and $j
```

In this case, taintedness can simply be propagated from the right to the left hand objects.

Address flow is associated with indirect reference through addresses. In the following sample, **$o** is dependent on **$i** (and **$table**):

```
$o = $table[$i];    // $o depends on $i
```

Control flow is associated with if-statement in high-level languages or conditional branches in binary codes. In the following sample, **$o** is dependent on **$i**:

```
if ($i == ' ')
  $o = '+';         // $o depends on $i
```

As described before, control flow is more difficult to track than data and address flows. DTPs

can find the dependent range of the variables appeared in the condition of if statement because it can see the block structure of the statement, which is explicitly specified in the source code.

On the other hand, it is very difficult for DIFTs to find that of a conditional branch. In order to do so, DIFTs must find the join point of the conditional branch, which is not explicitly specified in usual instruction-set architectures.

As far as we know, no DIFT can track control flows, and only a few DTP can track and propagate taint along with control flows.

5.2 Non-propagation policies

The output of a realistic application program is always a response to user input. Thus, a *perfect* DTP/DIFT, which can perfectly track all the kind of information flow described above, would mark all the output as tainted. Such DTP/DIFT is useless. Therefore, the *non-propagation policy*, is as important as how to propagate taint in particular in control flows. But as far as we know, all the current DTP/DIFTs define non-propagation policy on heuristics, such as about *sanitization* or *table reference*.

5.2.1 Sanitization

Most DTPs regard sanitized string as safe. Since sanitization is often performed by regular expression match and replace in script languages, most DTPs untaint strings which experience it. Some DTP untaint strings tested for the presence of particular (unsafe) characters. However, this is a well-known security hole.

5.2.2 Table reference

Most DTPs regard the values obtained from hash tables or arrays as safe, and do not propagate taint from the input to the output. Most script languages support hash table or array in such a form as follows:

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

Hash table retrieving is performed in the following manner:

(1) The input string is converted to a scalar index by a hash function.
(2) The chain indicated by the index is selected.
(3) Following the selected chain, the key string of one tuple in the chain is compared with the input string after another. If a key string matches the input string, the tuple is selected.
(4) Finally, the value string of the selected tuple is copied to the output string.

Most DTPs do not propagate taint through hash tables or arrays. In addition, some DTPs generalize the policy to string-to-scalar conversion, which hash functions perform[8),9)]. These DTPs regard all scalars as safe even if they are produced from tainted strings, because script injection attacks are performed finally by strings. And most DIFTs do not propagate taint along with address flows by default[4)], which prevents to propagate taint from the index to the values in hash table or array accesses.

6. SWIFT

This section presents the proposal of SWIFT. The goal of SWIFT proposal is to provide a high accuracy on detecting script injection attacks.

First of all, subsection 6.1 and 6.2 describes two key observations for introducing our proposal. Then, we describe the detection of string access in subsection 6.3. Finally, we present the taint information propagation technique of SWIFT in detail.

6.1 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*[10)].

As the example of SQL injection we describes in section 2. 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 example, 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.

In the example of section 2, it is very possible that the programmer may apply lower-case conversion to**$name** only because all the name of the products in the database is stored in lower-cases. If the programmer write lower-case conversion like figure 1(d), almost all the current DTPs untaint **$name** because it uses a translation table. In this case, however, such an attack is still possible even though **$name** is converted to lower-case (note that the SQL keywords are case-insensitive). Then, the current DTPs result in a false negative. So the substring corresponds to **$name** should

be always tainted even in non-attack cases. In the example of section 2, even if **ruby** is left tainted, the parser can distinguish it is not an attack.

The next subsection describes appropriate non-propagation rules when used with command parsing.

6.2 Radio-button and text-box operations

A *radio button* and a *text box* are two representative user interfaces on web pages. Radio buttons are used to choose one from some options. On the other hand, text boxes are used to get arbitrary strings, often with string conversions, such as case conversion or coding conversion.

As described in section 2, text boxes are unsafe to injection attacks. The programmer must carefully check the string entered through text boxes. Whereas radio buttons are considerably safe. Since the options of a radio button are provided by the programmer, the string that the user choose is necessarily under control of the programmer. It is practically impossible for attackers to attack through radio buttons.

The string operations in web applications can also be divided into radio-button and text-box.

6.2.1 Radio-button operations

figure 1(a) is a sample code of radio-button. As is the case of radio buttons, the value of the output **$ostr** is chosen from the strings given by the programmer according to the input **$i**. The programmer cannot predict the exact value of **$ostr**, but can completely predict the range of the value that **$ostr** will take. It can be said that the programmer has control on the output.

figure 1(b) is another implementation of figure 1(a). This is the same as hash tables described in section 5.2. In this case, the programmer also has control on the output. It is, however, not because hash tables are considered safe as described in section 5.2, but because the contents of the table has been given by the programmer.

6.2.2 Text-box operations

figure 1(c) is a sample code of text-box operation. Though the switch statement in the block of the for statement looks like the code in figure 1(a), it is actually of text-box. This is an inefficient implementation of lower-case conversion. As described in the previous subsection, the user can control the value of **$ostr** except that it is converted in lower-case, and it is still possible to attack through this operation. It cannot be said the programmer has control on the output.

Text-box operations include string copy and all kinds of string conversions such as case conversions explained above, or coding conversions. URL encode/decode and character code conver-

```
/* ... */
  switch ($i) {
  case 'A': $ostr = "alpha"; break;
  case 'B': $ostr = "beta";  break;
  /* ... */
  }
```

(a) Sample code of radio-button operation

```
$table['A'] = "alpha";
$table['B'] = "beta";
/* ... */
  $ostr = $table[$i];
```

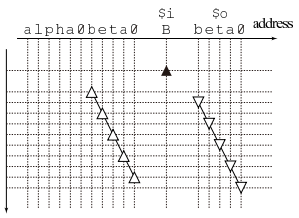
(b) Sample code of radio-button operation

```
for ($i = 0; $i < strlen($istr); $i++)
  switch ($istr[$i]) {
  case 'A': $ostr[$i] = 'a'; break;
  case 'B': $ostr[$i] = 'b'; break;
  /* ... */
  }
```

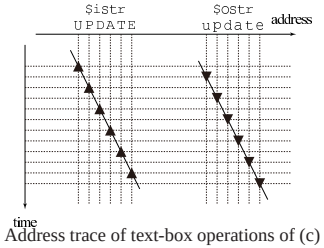
(c) Sample code of text-box operation

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

(d) Sample code of text-box operation



(e) Address trace of radio-button operations of (a) and (b)



(f) Address trace of text-box operations of (c) and (d)

Figure1 Radio-button and text-box operations and their address trace

sions are the most frequently used in web applications.

figure 1(d) looks like the radio-button operation of figure 1(b). In addition, the contents of the **$table** is also provided by the programmer. However, in fact it is just another implementation for figure 1(c) and of text-box.

6.2.3 Command parsing and string operations

In an application, strings travel on the route from the input to the output, experiencing one or more radio-button and/or text-box operations. Substring of the string can be considered under control of the programmer if it experiences at least one radio-button operation on the route.

Therefore, what we should do is to detect whether the operation that a substring experiences is radio-button or text-box, and propagate the taintedness of input string to output string if and only if the operation is detected as text-box.

Notice that this will propagate taintedness to the output even if it is not an attack. This is acceptable, since SWIFT is assumed to be used with command parsing described in the previous subsection. The parser will detect it is an attack or not correctly.

6.3 Algorithm of SWIFT

6.3.1 Address trace

As described in the previous subsection, the switch statements in figure 1(a) and 1(c) are almost the same except that the latter is included in the for statement. In order to distinguish the operation in figure 1(c) is of text-box, DTP has to know the switch statement is in the for statement and repeatedly executed producing a single string. It is difficult for conventional DTPs, which focus on each statement or instruction being executed as it did not grasp the overall situation.

SWIFT focus on memory access during program execution, and make use of address traces on the string operations. Figure 1(e) and 1(f) show two examples of address traces of these string operations. In these figures, the x-axis indicates the address, and the y-axis indicates the time. There are four types of triangles. Upward triangles △ and ▲ indicate load instructions to untaint and taint data, respectively. Downward triangles ▽ and ▼ indicate store instructions whose store value should be untainted and tainted, respectively. A group of triangles connected with a line indicates a string access. The load/store instructions that does not related to DTP are not drawn in these figures.

6.3.1.1 Basic address traces

Figure 1(e) corresponds to the sample code of the radio-button operation shown in figure 1(a).

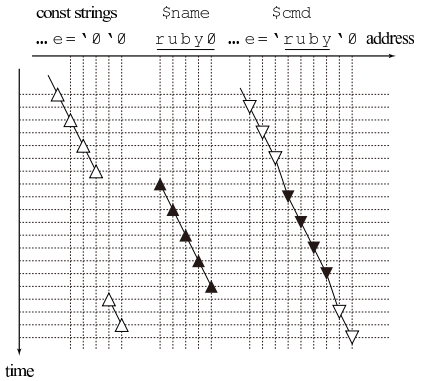


Figure2 Address trace of string concatenation of SQL injection

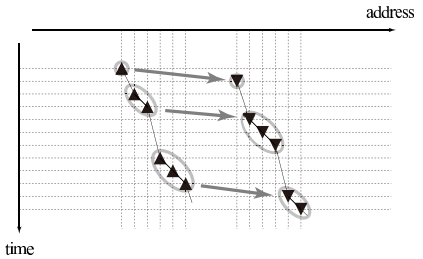


Figure3 Address trace of multi-byte string operation

The first tainted load indicates the load to input variable **$i**. In this case, the value of **$i** is **'B'**, then the constant string **"beta"** is copied to the output variable **$ostr**.

Figure 1(f) corresponds to the sample code of the text-box operation shown in figure 1(c). The taint input string **"UPDATE"** is converted to **"update"**.

In the both figures, the load instructions read strings and the store instructions write strings, the actions appear in an interleaved pattern. The obvious difference is that the loads are untainted in radio-button while tainted in text-box.

The sample code in figure 1(b)/1(d) are semantically the same as 1(a)/1(c), and their address traces also looks like 1(e)/1(f), respectively.

Figure 1(b) is of a hash table access. As shown in section 5.2.2, the hash table access starts with

hashing the input, and ends with copying the value of the selected tuple to the output. Figure 1(e) only shows the first taint load to the input to hash it, and the following string copy from the value string of the tuple to **$ostr**. The values of the tuples are untainted, since it has been copied from the untaint constant strings just in the reverse direction of figure 1(e).

Figure 1(d) uses a translation table. In the **$i**-th iteration, the **$i**-th character of **$istr** is loaded, then the value in the **$table** corresponding to the loaded character is loaded, and finally the value is stored to **$i**-th character of **$ostr**. Figure 1(f) only shows the first taint loads to **$istr**, and the last stores to **$ostr**, omitting intermediate untaint loads to **$table**. As shown in figure 1(f), it can be detected of text-box focusing on the taint loads to **$istr**, and the stores to **$ostr**.

So for script injection attacks detection, we should trace interleaving string reads and writes, and the taintedness of the read string should be propagated to the write string. And as the result, we can get the output of radio-button operations be untained, and the output of text-box operations be tainted.

6.3.1.2 Other examples

Figure 2 shows the address trace of SQL injection. In this case, the constant strings and user input **$name** are concatenated to produce **$cmd**. As shown in figure 3, the three substrings of **$cmd** should be tainted or untainted depending on the source strings are taint or untaint. And here, the string **”ruby”** could be tainted because the parser will detect it is not an attack.

Figure 3 shows a copy processing of multi-byte string operation. Web applications often deal with multi-byte characters, such as URL coding or non-ASCII character sets.

6.3.2 String access detection

6.3.2.1 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. Figure4 shows an example of an interleaving pair. This figure shows a address trace of base64_encode. The x and y axis show memory addresses and time. 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**.

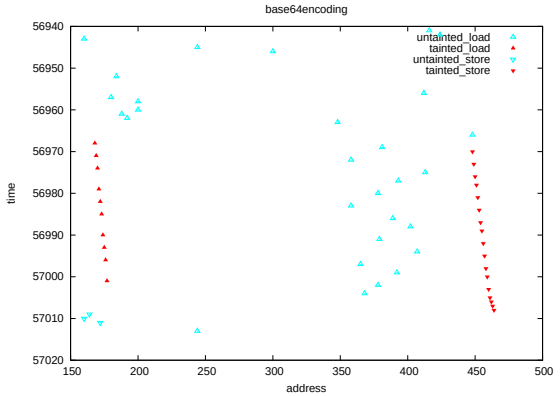


Figure4 Example of interleaving pair (base64_encode)

6.3.2.2 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.

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*.

6.3.2.3 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.

6.3.2.4 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.

6.3.2.5 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.

7. 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 scripting language that is designed for server-side web development. By implementing SWIFT to PHP, coverage is limited for PHP, but a highly accurate DTP can be made in the range of taint-support PHP. And it is easy for SWIFT to be familiarized.

7.1 PHP interpreter

We describe the interpreter of PHP.

Figure 5 shows a sample PHP script. We refer functions users defines as PHP user-defined functions. In this script, caselow() is a PHP user-defined function. Strlen() in caselow() is a PHP built-in function.

A script of PHP is compiled to the intermediate language named opcode by a runtime compiler and is executed. Figure 6 shows a dump of opcode. PHP user-defined functionsand PHP built-in functionsare called by opcode named DO_FCALL.

PHP interpreter is written in C. We refer functions that execute each opcode as Opcode functions. We refer functions that execute each PHP built-in functionas Zif functions.

All variables in a PHP script are stored in the structure named zval after they are compiled. Zval is Figure 7 Zvalue_value in the zval is Figure 8 When a string in a script is stored in zval, a pointer to the string is acquired by using macros, such as Z_STRVAL(), Z_STRVAL_P() and Z_STRVAL_PP(), in the Opcode functions. The argument of Z_STRVAL is zval, and the rest of the macros take the pointer to zval as their arguments.

```
<?php
$str = "toihiroshi";
$lower = caselow($str);
echo $lower;

function caselow($str){
  $n = strlen($str);
  for ($i = 0; $i < $n; $i++){
                    switch ($str[$i]){
                    case'A':        $dst .= 'a';        break;
                    case'B':        $dst .= 'b';        break;

/*.....*/

                    case'Z':        $dst .= 'z';        break;
                    default:        $dst .=$str[$i]; break;
                    }
  }
  return $dst;
}
?>
```

Figure5 Sample PHP script

7.2 Acquisition of memory addresses

We explain the acquisition of 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 on the source code of Opcode functions and Zif functions.

7.2.1 Opecode functions

String access is done by using the macros mentioned above in Opcode functions, 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.

```
function name:  (null)
number of ops:  7
compiled vars:  !0 = $str, !1 = $lower
line    # op                    fetch       ext  return  operands
-------------------------------------------------------------------------------
  2     0  NOP
 41     1  ASSIGN                                      !0, 'toihiroshi'
 42     2  SEND_VAR                                    !0
        3  DO_FCALL                              1     'caselow'
        4  ASSIGN                                      !1, $1
 43     5  ECHO                                        !1
 45     6  RETURN                                      1

function name:  caselow
number of ops:  151
compiled vars:  !0 = $str, !1 = $n, !2 = $i, !3 = $dst
line    # op                    fetch       ext  return  operands
-------------------------------------------------------------------------------
  2     0  RECV                                        1
  3     1  SEND_VAR                                    !0
        2  DO_FCALL                              1     'strlen'
        3  ASSIGN                                      !1, $0
/*.....................................................................*/
  6    10  FETCH_DIM_R                                 $5     !0, !2
  7    11  CASE                                  ~6    $5, 'A'
       12  JMPZ                                        ~6, ->16
       13  ASSIGN_CONCAT                                   !3, 'a'
       14  BRK                                   1
/*.....................................................................*/

 32   135* JMP                                          ->138
      136  CASE                                  ~6    $5, 'Z'
      137  JMPZ                                        ~6, ->141
      138  ASSIGN_CONCAT                                   !3, 'z'
      139  BRK                                   1
/*.....................................................................*/
 36   148  JMP                                          ->7
 37   149  RETURN                                       !3
 38   150* RETURN                                       null
```

Figure6 Dump of opcode

One case is that functions that manage the memory of the interpreter take the macros as thier arguments. The function to which we should pay attention is five of the following: estrdup(), estrndup(), estrndup_rel(), erealloc(), memcpy(). Memcpy() is C built-in library function. The e*()s are defined on the interpreter, and they use memcpy() internaly. For example, we can find source code as below:

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

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

Figure7 Zval

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

Figure8 Zvalue_value

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.

7.2.2 Zif functions

There are about a hundred Zif functions from which we have to get memory addresses. For example, zif_urlencode(), zif_base64_encode(), zif_ereg_replace(). Because only pointers to char are passed to Zif functions, we must read the source code and get memory addresses.

8. Evaluation

8.1 Environment

We implemented SWIFT to PHP-5.3.1. As for taint-support PHP, we used PHP-taint 20080622 package.

We set the environment as below: Ubuntu 9.04, Apache 2.2.14, Mysql 5.1.37.

8.2 String operations

Table 1 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 opera-

	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)	tolower (switch-statement)			√	

FN : false negative FP : false positive

Table1 Results of string operation

tions, 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 lowercase conversions code, shown in figure 1(c). Though the function is the same as (5)strtolower(), 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 taint 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.

8.3 Real-world web applications

We executed eight 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. These applications use some input variables as an argument without validation or even any string operations to them. We made Script injection attacks such as Cross Site Scripting (XSS),

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

Table2 Results of web applications

directory traversal, and SQL injection according to the exploit code. As summarized in Table 2, SWIFT caused no false positives or negatives. But PHP-taint produced false negatives.

8.4 Accelerator

we evaluated PHP-SWIFT with a PHP accelerator. A PHP accelerator is an extension designed to speed up execution time of software applications written using PHP. Most PHP accelerators use opcode caches. Opcode caches work by caching compiled codes of a PHP script (opcode) in shared memory to avoid the overhead of parsing and compiling source code on each request. Some users and enterprises introduce accelerators in order to increase a speed of PHP code.

We used an eAccelerator[?)] as an accelerator. The eAccelerator is one of free accelerators. We confirmed PHP-SWIFT worked correctly with the eAccelerator.

9. Conclusion

In this paper, we implemented String-Wise Information Flow Tracking, SWIFT, to PHP. SWIFTis a completely different approach from conventional DTP/DIFTs. In order to detect script injection attacks precisely, SWIFT observes memory accesses of a target programs, detects text-box string operations and propagates taint through them. Since SWIFT only uses address traces of a program, it can be implemented both on script language interpreters and on processors.

Vol.2010-OS-115 No.4
2010/8/3

We implemented SWIFT to PHP and compared the accuracy with taint-support PHP. PHP-SWIFT can correctly propagate taint for typical string operations and real-world web applications with known vulnerabilities, while PHP-taint don't. Additionally, we confirmed PHP-SWIFT worked correctly with the eAccelerator.

We are going to implement SWIFT to all PHP built-in functions. We plan to distribute PHP-SWIFT.

References

1) Allen, J.: Perl Version 5.8.8 Documentation - Perlsec, http://perldoc.perl.org/perlsec.pdf (2006).
2) Chen, H., Wu, X., Yuan, L., Zang, B., chung Yew, P. and Chong, F.T.: From Speculation to Security: Practical and Efficient Information Flow Tracking Using Speculative Hardware, *Int'l Symp. on Computer Architecture*, pp.401–412 (2008).
3) Christey, S. and Martin, R.A.: Vulnerability Type Distributions in CVE, http://cve.mitre.org/docs/vuln-trends/ (2007).
4) Dalton, M., Kannan, H. and Kozyrakis, C.: Raksha: A Flexible Information Flow Architecture for Software Security, *34th Int'l Symp. on Computer Architecture*, pp.482–493 (2007).
5) Haldar, V., Chandra, D. and Franz, M.: Dynamic Taint Propagation for Java, *21st Annual Computer Security Applications Conf.*, pp.303–311 (2005).
6) Livshits, B., Martin, M. and Lam, M.S.: SecuriFly: Runtime Protection and Recovery from Web Application Vulnerabilities, *Tech. Rep., Stanford Univ.* (2006).
7) Nanda, S., Lam, L.-C. and cker Chiueh, T.: Dynamic Multi-Process Information Flow Tracking for Web Application Security, *8th Int'l Middleware Conf.* (2007).
8) 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.*, pp. 295–307 (2005).
9) 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).
10) Su, Z. and Wassermann, G.: The Essence of Command Injection Attacks in Web Applications, *33rd Symp. on Principles of Programming Languages* (2006).
11) 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*, pp.85–96 (2004).
12) 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).