An Auto-tuning Sanitizing System for Mitigating Injection Flaws

Jan-Min Chen
(Corresponding author: Jan-Min Chen)
Department of Information Management, Yu Da University of Science and Technology
No. 168, Hsueh-fu Rd., Tanwen Village, Chaochiao Township, Miaoli County 361, Taiwan, R.O.C.
(Email: ydjames@ydu.edu.tw)
(Received July 8, 2013; revised and accepted May 20, 2014)

Abstract

Injection attacks are dangerous and ubiquitous, contributing enormously to some of the most elaborate Web hacks. Enforcing proper input validation is an effective countermeasure to improve injection flaws. Unless a web application has a strong, centralized mechanism for validating all input from HTTP requests, injection flaws are very likely to exist. However, improper constraining rules may induce some detection error. False negatives may render security risks and false positives will cause improper limits of input characters. In this paper, we design an auto-tuning system to help validating input for each vulnerable injection point. A proper validation rule can be automatically generated through an auto-tuning mechanism. The experimental results show that the system can effectively protect against injection attacks and lower false positives while compared with traditional methods.

Keywords: Constraining rule, content filtering, detection accuracy, injection flaws, input validation

1 Introduction

Injection attacks can be very easy to discover and exploit. Hackers take advantage of a weakness in the Web application design to intentionally insert some extra characters in input data to bypass or modify the originally intended functionality of the program. The consequences can run the entire range of severity, from trivial to complete system compromise or destruction. Many application's security vulnerabilities result from generic injection problems. Examples of such vulnerabilities are SQL injection, Shell injection and Cross site scripting (XSS). Enforcing input validation is an effective countermeasure to protect against injection attacks. Traditional methods usually use a generic constraining rule to strictly sanitize all input. It may cause improper limits of input characters because of some false positives.

There are numbers of the researches related to protect Web site against injection attacks: Huang et al. had developed a WebSSARI and a WAVES [8]. The Open Web Application Security Project (OWASP) had launched a WebScarab project [16]. The other available commercial scanners also included IBM Rational's AppScan and SPI Dynamics' WebInspect [9, 23]. Above-mentioned tools just focus on finding Web application flaws. Once web application vulnerabilities have been identified, the ultimate solution is to fix the vulnerabilities in the web application source code itself. However, this can render intrusion risks because proper vulnerability fixing often requires doing something else such as testing, supports coming from third parties and vendors of multiple software components. Thus, some solutions had been proposed to protect against attacks before fixing flaws. Sanctum Inc. provided an AppShield adopting Security Gateway to prevent application-level attacks [22]. Some advanced firewalls also incorporated deep packet inspection technologies for filtering application-level traffic [3]. We had proposed a fixing tool that can be used to improve injection flaws [11]. It can produce proper input validation functions related to the source codes of applications. Next an enhanced prototype adopting a security gateway in front of web server to sanitize malicious input had been proposed solving the problem as source code may not be modified [13]. The above two methods both use a generic constraining rule to validate input, so false positive will be troublesome. Recently, Web application firewalls (WAF) is a popular solution to be used to create an external security layer to improve security, detection, and prevention of attacks before they hit web applications [10, 15, 17]. For WAF, the sanitizing mechanism is a critical technique. However traditional validating methods are susceptible to error because of using single constraining rules to sanitize all input. Lately, we had proposed a heuristic mechanism that can automatically generate proper validation rules based on each vulnerable injection point. The method had been primarily proved both guarantee security (false negatives) and convenience (false positives) [2].

In this paper, we create a sanitizing system to help validating input. For each vulnerable injection point, a proper validation rule can be automatically generated and adjusted itself to new injection attacks through an auto-tuning mechanism. Thus the system can both guarantee better detection accuracy compared with other constraining strategies and better effectiveness to protect against new injection attacks.

The main contributions of this paper are summarized below:

1) It has good "scalability" when applied to Web site growth or new attack patterns because it integrates an injection vulnerability analyzer (finding target), an injection pattern generator (exploitation), and a constraining rule generator (prevention) into an auto-tuning bastion.

2) It proposes an auto-tuning mechanism to effectively improve the detection accuracy of the signature-based content filtering techniques.

3) It designs a system to automatically protect against new injection flaws through the seamless delivery of new constraining rule.

The rest of this article is organized as follows: The second section surveys a number of works relevant to improve injection flaws. In third section, we describe our technical details of auto-tuning sanitizing agent. The system implementation is shown in fourth section and its effectiveness is evaluated in fifth section. The last section concludes the whole paper.

2 The Works Relevant to Improve Injection Flaws

2.1 Injection Flaws

Injection flaws allow attackers to relay malicious code through a web application to another system. The attacker can inject special (meta) characters, malicious commands, or command modifiers into the information and the web application will blindly pass these on to the external system for execution. These attacks include calls to the operating system via system calls, the use of external programs via shell commands, as well as calls to backend databases via SQL [18].

SQL injection is a type of security exploit in which the attacker adds SQL statements through a web application's input fields or hidden parameters to gain access to resources or make changes to data. It is a particularly widespread and dangerous form of injection. It is an attack technique used to exploit web sites that construct SQL statements from user-supply input. SQL injection is a serious vulnerability, which can be found in any environment with an SQL back-end database (Microsoft SQL Server, Oracle, Access, and so on) and used to steal information from a database from which the data would normally not be available and to gain access to host computers through the database engine. As with SQL injection, XSS is also associated with undesired data flow. XSS exploit vulnerabilities in Web page validation by injecting client-side script code. The script code embeds itself in response data, which is sent back to an unsuspecting user. The user's browser then runs the script code. Because the browser downloads the script code from a trusted site, the browser has no way of recognizing that the code is not legitimate. One of the most serious examples of a XSS attack occurs when an attacker writes script to retrieve the authentication cookie that provides access to a trusted site and then posts the cookie to a Web address known to the attacker. This enables the attacker to spoof the legitimate user's identity and gain illicit access to the Web site.

2.2 Content Filtering

There are two strategies are typically employed in content filtering: signature-based and heuristic-based. Simple signature-based detection is an effective and computationally efficient method to detect viruses, but it does have a couple of shortcomings. Signature-based detection involves searching for known patterns of data within executable code. However, it is possible for a computer to be infected with new malware for which no signature is yet known. One type of heuristic approach is intended to overcome the shortcoming. Heuristic-based detection, like malicious activity detection, can be used to identify unknown viruses. Because viruses tend to perform certain actions that legitimate programs do not, they can usually be identified by those actions. If heuristic detection was employed, success depends on achieving the right balance between false positives and false negatives. Due to the existence of the possibility of false positives and false negatives, the identification process is subject to human assistance which may include user decisions, but also analysis from an expert of the antivirus software company [1].

Signature-based intrusion detection system such as the popular Snort program is typically configured with a set of rules to detect popular attack patterns. These rules look almost exactly like firewall rule sets in that patterns can be specified on packet header fields using the usual flexibility of specifying prefixes, wildcarded fields, and port ranges [5]. However, signature detection systems go one step beyond packet filters in complexity by also allowing an arbitrary string that can appear anywhere in the packet payload. String matching in packet content, is also of interest to many applications that make use of content-based forwarding. Radwan et al. show a new implementation of a gateway capable of applying content-based security on attachments of messages, where a single gateway serves several web servers in a web farm [24].

2.3 Constraining Input

Input validation is a secure input handling way for verifying user input to ensure that input is safe prior to use. In general, it checks the user input through constraining rule based on two types of security model (a whitelist and a blacklist). Validation based on whitelist can ensure that all requests are denied unless specifically allowed. A whitelist is a set of all allowed items. The list may involve setting character sets, type, length, format, and range. On the other hand, a blacklist defines what is disallowed, while implicitly allowing everything else.

The benefit of using a whitelist is that new attacks, not anticipated by the developer, will be prevented. A generic solution may not be easily implemented but we can know that is acceptable input data for application program in a localized way. It is much easier to validate data for known valid patterns but it may induce more false positives. On the other hand, a blacklist can clearly dictates that you should specify the characteristics of input that will be denied. Ultimately, however, you'll never be quite sure that you've addressed everything through the blacklist. That is to say, it is an unrealistic idea assuming that all the variations of malicious injection had been known. The blacklist may be quite tempting when you're trying to prevent an attack on your site. However, it allows more abundant input data than a whitelist. In summary, a blacklist often can guarantee fewer false positives than a whitelist but it may induce more false negatives.

2.4 Improving Injection Flaws

Once injection flaws have been identified, the ultimate solution is to fix the vulnerabilities in the web application source code itself. However, this can't be reachable immediately because proper vulnerability fixing often requires doing something else such as supports coming from third parties and vendors of multiple software components. Thus, some solutions had been proposed instead to improve injection flaws before fixing the vulnerabilities.

The Open Web Application Security Project recommends that a thorough validation of any input data needs to be made in order to ensure that the data does not contain any malicious content [16]. SPI Dynamics also suggest using regular expressions for sanitizing data before it is executed by a back-end database [23]. There has been other research into improving injection flaws. Salem et al. had outlined an intercepting filter approach aimed at increasing the security and reliability of web applications by eliminating injection flaw exploitations. The use of filter components, in conjunction with the Intercepting Filter design pattern, can be used to sanitize HTTP Request information before it is ever processed by the web application and had been carried out on Java and .NET based platforms [21]. DOME uses a filter that looks for and marks the locations of system calls and then watches the result of the execution of the actual code [19]. Halfond et al. had proposed a technique that uses a program to automatically build a model of the legitimate queries that could be generated by the application [6].

3 Protecting Against Injection Attacks

An injection flaw is the result of an invalidated input and thus, proper input validation is an effective countermeasure for protecting against injection attacks. In particular, some input validation programs are poorly written, lacking even the most basic security procedures for sanitizing input. Furthermore, some legacy applications may not be able to modify the source of such components. Currently, a WAF is a common solution that can be used in addition to the protected Web site to prevent an immediate injection attack. Although a WAF is language independent and requires no modification to the application source code, it may induce false recognition due to the use of a generic constraining rule.

3.1 Sanitizing Agent

To help performing proper input validation is an effective countermeasure for mitigating injection flaws. It can be achieved by a two steps approach: first, to find all vulnerable injection points and second, to automatically and accurately validate input.

Injection flaws can be found via the Input Validation Testing (IVT). Here the IVT is defined as choosing proper test case that attempt to show the presence or absence of specific errors pertaining to input data [7]. A Web application vulnerability scanner is a common IVT tool. We also had proposed a feasible method for performing IVT. The method not only uncovers vulnerability but also ensures the location where the vulnerability occurs [12]. Upon completion of processes of Web crawling and analysis, an injection point list can be created and filled in the fields of URL and parameter with the result of analysis. And then, IVT will be launched to uncover different vulnerabilities according to the injection point list. The completion of IVT allows us to fill in flaw type field of an injection point list with flaw name to enable the recording of flaws of each injection point.

Next, the task to automatically assist on validating input for each vulnerable injection point is achieved by a sanitizing agent. The sanitizing agent can help validating input via meta-programs. The meta-programs can be translated by a code generator. Each parameter of the same URL in the vulnerable injection point table can generate a meta-program to constrain inputs. For example, if there are some records having the content of a URL field as 'verify.php' in the vulnerable injection point table, our mechanism can automatically generate a meta-program named 'verify.php' to help sanitizing the http requests which surfing destination is 'verify.php'. The Algorithm1 is used for generating a meta-program and the code snippet of the meta-program is presented in Table 1. It needs

to be noted that the italic words should be replaced by a parameter field in the vulnerable injection point table and the constraining rule while generating a meta-program. The constraining rule can be gotten through looking up the constraining rule table according to the flaw type field in the vulnerable injection point list. The meta-program can perform input validation instead of the Web applications having injection flaws. However the meta-program only embodies a sanitizing functionality and so it can't replace the initial program. Thus, after completing the sanitizing procedure, the http request needs to be redirected to original program to obtain prime service.

Algorithm 1 Generating a meta-program

```
//A algorithm for generating a meta-program named as url
OPEN a vulnerable injection point table
GET url
FOR EACH distinct url in a vulnerable injection point table
  FOR EACH parameter having same url
//To generate a program segment of validating input for each vulnerable injection point
   GET constraining rule, parameter's value
   STORE parameter's value to parameterValue // input data
//To search the parameterValue for the number of times of match to the regular expression given in constraining rule
   FOR EACH parameterValue // validation logic begin
    COUNT the number of times of match to the regular expression given in con-straining rule
    STORE the number of times of match to Counter
   END FOR
   IF (Counter > legalSpecialCharCount)
    ECHO error message
    EXIT
   END IF
   SESSION parameter's value // validation logic end
   FILEWRITER url // append the validation logic
  END FOR
  REDIRECT url
END FOR
```

3.2 Generating Ideal Constraining Rules

In general, the constraining rule was based on a generic whitelist or blacklist. A generic whitelist usually only allows case-sensitive alphanumeric characters. A generic blacklist always does its best to include all possible malicious characters to guarantee same false negative. A blacklist allows more abundant input data than a whitelist so validation based on a blacklist can cause fewer false positives than a whitelist. However, it may induce more false negatives. In general case, a configurable set of malicious characters is used to reject the input but it is an unrealistic idea assuming that all the variations of malicious injection had been known. Therefore a validation based on a blacklist should guarantee acceptable false negative, and then do everything possible to reduce false positive. The generic constraining rule can guarantee security, but it may cause more false positives, causing inconvenient because of improper limitation of input characters. Thus, we need an intelligent method for gathering necessary characters in a blacklist to generate an ideal constraining rule according to the actual situation. While inspecting some injection attack strings, we find that most special (not case-sensitive alphanumeric) characters in the strings don't appear alone. For the special characters appearing in an injection string, just only one of them is required to be added in a blacklist. Thus we think that a better method is only put necessary special characters in a blacklist.

Table 1: Example of a simple meta-program

```
<? // a simple meta-program
  $id0=$_POST['myusername'];
  $pattern0="/[=;_'>%<\(@:&\\-\|!\.\+\/]/";
  $i = 0;
  while((preg_match($pattern0, $id0, $matches))
    && $i < 2 )
    {
        $i++;
        $temp = preg_split("/[$matches[0]]/",
          $id0);
        $id0 = implode($temp);
    }
    if ($i >= 2)
    {
        echo "illegal character detected";
        exit;
    }
  $_SESSION["myusername"] =$id0;
  redirect ($url);
?>
```

We think that the necessary special character is which having most appearing rate in comparison with other special characters in an injection string. According to the clue, we propose the Algorithm2 for choosing the special characters really having to be added in the blacklist. The algorithm can be used to generate an ideal constraining rule according to various types of injection patterns. Thus the ideal constraining rule can be used in the meta-program to not only guarantee same false negative but also reduce more false positives in comparison with using a generic blacklist while performing input validation.

Algorithm 2 Generating a ideal constraining rule

```
FOR EACH injectionString(i)
  Let candidateString(i) = all specialCharacters in an injection string
  //specialCharacter i.e. not case-sensitive alphanumeric
  Remove same characters from candidateString(i)
  Let specialCharacter (i, j) = the specialCharacter had appeared in candidateString(i)
  For EACH specialCharacter (i, j)
    Count the amount of the specialCharacter appearing in all candidateStrings and STORE to appearRate(i, j)
  END FOR
END FOR
FOR EACH candidateString(i)
  SORT the specialCharacter(i, j) in the candidateString by appearRate (i, j) into descending order
  IF (the first character in the candidateString can't be found in a blackList)
    ADD the character to the blackList
  END IF
END FOR
```

3.3 An Auto-tuning Mechanism

Generating an ideal constraining rule is a critical technique for mitigating injection flaws. Ideal sanitizing input depends on achieving the right balance between false positives and false negatives. Traditional methods usually use a comprehensive constraining rule to strictly sanitize input. It may cause improper limits of input characters because of some false positives. A looser rule may lower false positives than a generic rule but it may induce false negatives. False negatives may render security risks. Thus, from a security defense viewpoint, the least false negatives should have a higher priority than the false positive and it follows that, in general, the false negative is zero. Due to the existence of the possibility of false positives and false negatives, the identification process is subject to human assistance which may include user decisions.

Although a blacklist is likely to support more elastic input data than a whitelist, it may induce more false negatives. The blacklist may be quite tempting so it is an unrealistic idea assuming that all the variations of malicious injection had been known. Thus a validation based on blacklist should prefer assuring of acceptable false negative, and do everything possible to reduce false positive. Our auto-tuning approach is intended to automatically achieve the right balance between false positives and false negatives. It guarantees same number of false negatives as well as reduces more false positives while sanitizing inputs.

False positives may cause improper limits of input characters. There are some normal input including special characters such as compound name (jan-min including "−") and domain name (www.ydu.edu.tw including "."). Adding these special characters in the blacklist must induce many false positives, on the contrary, removing these special characters must render many false negatives. While inspecting some malicious injection strings, we find that most special characters in the strings don't appear alone. Thus we can pretend that it is a normal string if only one type of special character appearing in an injection string. In the algorithm1, generating a meta-program, we can let the legalSpecialCharCount equal to 1 for effectively lowering false positives.

The known malicious injection string must be detected and removed by the sanitizing agent. That is to say, there are normal data and new malicious injection strings will be kept in the Web access log of the protected Web server. The new malicious injection strings can be quickly filtered and categorized according to attack types and then add to the testing pattern table. Finally, new constraining rules can be generated to protect against new injection attack. Thus, the auto-tuning mechanism can help generating ideal constraining rules to achieve the right balance between false positives and false negatives.

4 System Implementation

To verify the effectiveness of our scheme, we implement an auto-tuning sanitizing system and present its architecture diagram and interactions between each component in Figure 1. The system consists of three main components: sanitizing agent, injection vulnerability analyzer and injection pattern generator. The injection vulnerability analyzer is responsible for finding injection flaws and generating a vulnerable injection point table [13]. The injection pattern generator is dedicated to organizing new injection patterns by analyzing the navigational information kept in the Web access log. The sanitizing agent is capable of help validating input. The sanitizing agent is allocated in front of the protected Web servers. All HTTP\HTTPS requests to the protected Web servers are routed through the sanitizer that can either deal with the request itself or pass the request partially to Web servers. After passing all check, the requests are forwarded to the Web server. The sanitizing agent deals with both the requests coming from client and the response pages coming from the Web server and then forward to server\clients. It can help validating input via meta-programs. Any malicious injections must be blocked by the sanitizing agent. The meta-programs can be translated by a code generator named translator. Each parameter of the same URL in the vulnerable injection point table can generate a meta-program to constrain inputs. Furthermore, the meta-program only has sanitizing functionality so it can't replace original Web application. Thus after completing constraining work, the http request needs to be redirected to original program to obtain prime service. In general, the known malicious injection string must be detected and removed by the sanitizing agent. That is to say, there are only normal data and new malicious injection strings will be kept in the

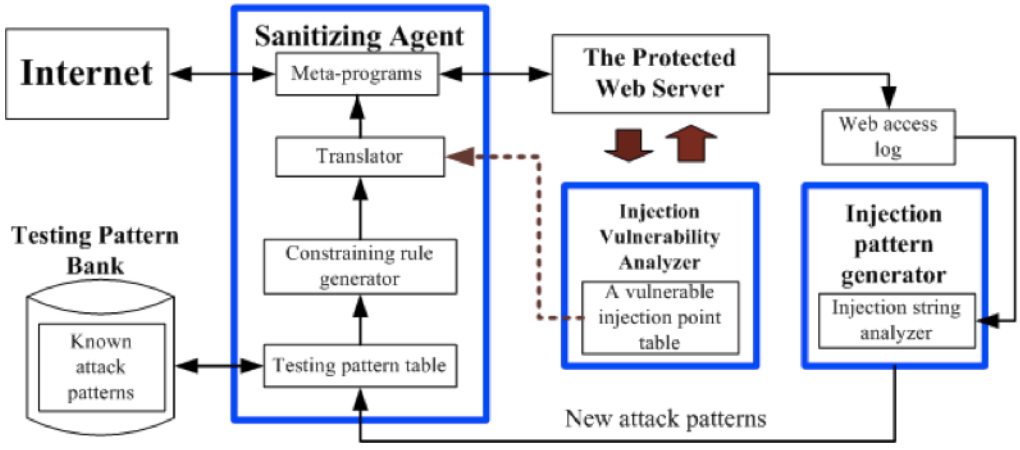

Figure 1: A architecture diagram of the auto-tuning sanitizing system

Web access log of the protected Web server. The injection pattern generator can analyze the Web access log and generate various injection patterns. These patterns can be categorized according to attack types and then kept in the testing pattern table. The constraining rule generator is capable of generating various ideal constraining rules according to testing pattern table. The detection accuracy of a sanitizing agent is dependent on constraining rules. The constraining rule generator may automatically organize new constraining rules while new injection patterns added in the testing pattern table. Thus the sanitizing rule can automatically adjust itself to constrain new malicious injection.

In Figure 1, the solid lines show the process of the auto-tuning mechanism. The sanitizing system can achieve the right balance between false positives and false negatives through auto-tuning mechanism. A false positive occurs when the normal input data is mistakenly blocked by a sanitizer, while a false negative occurs when the sanitizer cannot constrain a malicious injection. From a security defense viewpoint, the least false negative should have a higher priority than the false positive and it follows that, in general, the false negative is zero. A looser constraining rule may induce more false negatives. Thus some malicious injection strings will pass sanitizer and are kept in Web access log. These injection strings will be translate to new injection patterns by the injection pattern generator and then sent to testing pattern table. Next the constraining rule generator will organize stricter constraining rule to lower false negatives. Above processes will recursively go on until none of false negative. Therefore the auto-tuning sanitizing system can automatically adjust itself to effectively protect against new injection attacks.

We created an experimental website having SQL-injection and XSS vulnerabilities. The experimental website included 13 Web pages, 47 injection points, and 11 vulnerable injection points. We used the experimental website to assist in fine-tuning the detection accuracy of the injection vulnerability analyzer. To emphasize the importance of individual constraining rules, some injection points have been designated as special cases having only one specific flaw. For example, after the vulnerability assessment, we discovered that the third injection point of the experimental Web site only has XSS injection vulnerability and the fourth injection point has SQL vulnerability. This is then considered as a false positive if some pattern designed to locate SQL injection is filtered on the third injection point. We also chose six web applications from the National Vulnerability Database to enrich the evaluation of the injection vulnerability analyzer and the auto-tuning system [14]. After the injection vulnerability analyzer finishes the process, the detailed information about all the programs used in protected the Web site is presented in Table 2.

All primary tests were performed on an experimental Website including some client-side Web pages and vulnerable Web applications. The tests relevant to verifying the effectiveness of the sanitizer were divided into two phases. In the first phase, the injection vulnerability analyzer began to directly inspect vulnerability to create a vulnerable injection point table. In the second phase, at first the meta-program in the sanitizing agent adopted a looser constraining rule to verify the effectiveness of the auto-tuning mechanism. All known injection attack patterns were used as input. After completing the auto-tuning process, the false negative of the system is zero and the false positive is at strict minimum.

5 Experimental Evaluation

The effectiveness of the auto-tuning sanitizing system should be verified via two phases. One is to evaluate the detection accuracy of the system and next to show the auto-tuning mechanism can automatically adjust constraining rule to effectively sanitizing new injection patterns. There are none standard experimental data for verifying the performance of the system. In order to be sure of specific vulnerability, we need to create a testing

Table 2: The detail information about all programs used in protected website

Application Name/CVE#	Web page amount	Injection points amount	Vulnerable injection point amount	Vulnerable page Name	Flaw type
Experimental program	13	47	11	bbs.php	SQL injection/XSS
CVE-2010-0122	72	324	5	add_user.php	SQL injection
CVE-2009-4669	748	2947	2	Login.php	SQL injection
CVE-2009-4595	30	22	2	index.php	SQL injection
CVE-2010-1742	13	15	1	projects.php	XSS
CVE-2009-4456	9	10	1	news_detail.php	SQL injection
CVE-2009-3716	7	8	2	admin_login.php	SQL injection

Table 3: The constraining rule for various testing pattern sub-banks

Name	Description	constraining Rule_Name	Rule_Expression	String length of Rule_Expression
Testing bank 1	Sql_pattern_basic	Sanitizing_Sql_Rule 1	=;’	3
Testing bank 2	Sql_pattern_rich	Sanitizing_Sql_Rule 2	=%>(-*\’@:	10
Testing bank 3	XSS_pattern_basic	Sanitizing_XSS_Rule 1	<%	2
Testing bank 4	XSS_pattern_rich	Sanitizing_XSS_Rule 2	<%&(/=;\	8
Testing bank 5	Sql_pattern_rich & XSS__pattern_rich	Sanitizing_Sql&XSS_ Rule	’%>;-,=_/<(@:&	14
Testing bank 6	All malicious injection patterns	Sanitizing_all_Rule	=;_’>%/<(@:&\-|!.+	18

pattern bank including various testing patterns, relevant vulnerability types and comments. Each testing pattern should be meticulously designed to get expected output which can be clearly identified. AT first we had collected 1000 experimental data in a testing pattern bank. They came from access logs of websites which were scanned by Web vulnerability scanners or Web sites describing cheat sheet about injection attack [4, 20]. In Table 3, we present some testing pattern sub-banks for example. Each sub-bank is a part of the testing pattern bank and composed of patterns having specific types of vulnerability. The purpose of generating various sub-banks is to generate different constraining rules for specific injection flaws. For example, the Testing Bank1 only includes some popular SQL injection strings (10 classic patterns) and the Testing Bank2 put all SQL injection strings (453 patterns) together. The Testing Bank3 only includes some popular XSS injection strings (10 classic patterns) and the Testing Bank2 put all XSS injection strings (350 patterns) together. Others are the other types of malicious injection strings. In Table 3, we can find that the length of Rule_Expression of the Sanitizing Sql_Rule 2 is bigger than the length of Rule_Expression of the Sanitizing Sql_Rule 1 and the length of Rule_Expression of the Sanitizing all_Rule is biggest. It implies that the length of Rule_Expression may depend on the quantity and types of the injection strings.

We try to show that the detection accuracy of the sanitizer is dependent on the constraining rule. That is to say, the ideal constraining rules produced by the sanitizer can lower errors (false positives and false negatives) in comparison with the generic rules (generic whitelist or generic blacklist). From a security defense viewpoint, the least false negatives should have a higher priority than the false positive and it follows that, in general, the false negative is zero. In general, a useful sanitizing method must ensure that the false negative is zero. Thus the effectiveness of various sanitizing methods can be simplified to which having minimum false positive. A false positive occurs while these normal strings are mistakenly limited by a constraining rule. In order to be sure of false positive, we need to add some normal input strings (10 classic patterns) including at least one special character that appears only the first time in a testing pattern bank. All relevant parts of the testing results are presented as follows.

Table 4 and Table 5 show the amount of false negative and false positive induced by different constraining rules. A more flexible rule may induce less false positives than a generic rule, made apparent by the fact that the injection point only has single one specific flaw. For example, Table 4 shows that validation of using Sanitizing_ Sql_Rule 1 will only render less false positives in comparison with using Sanitizing_Sql_Rule 2 for the CVE-2009-4456.

From the principle of validating input and the experimental results we can conclude the following rules.

Table 4: The partial results of system training and detection experiment

CVE #	Program Name	Threat type	Sql_Rule 1+ #	Sql_Rule 1− #	Sql_Rule 2+ #	Sql_Rule 2− #
Experimental Web site	TestingInjection	SQL/XSS	1	1	2	1
CVE-2010-0122	timeclocksoftware	SQL	1	1	2	0
CVE-2009-4669	RoomPHPlanning	SQL	1	1	2	0
CVE-2009-4595	PHP Inventory	SQL	1	1	2	0
CVE-2009-1742	Scratcher	XSS	N/A	N/A	N/A	N/A
CVE-2009-4456	Green Desktiny	SQL	1	0	2	0
CVE-2009-3716	MCshoutbox	SQL	1	1	2	0

PS:
Sql_Rule 1+ #: the false positive amount of rending by the Sanitizing_SQL Rule 1
Sql_Rule 1− #: the false negative amount of rending by the Sanitizing_SQL Rule 1

Table 5: The partial results of system training and detection experiment

CVE #	Threat type	Sql& XSSRule+ #	Sql& XSS_Rule− #	All_Rule+ #	All_Rule− #
Experimental Web site	SQL/XSS	2	0	2	0
CVE-2010-0122	SQL injection	2	0	2	0
CVE-2009-4669	SQL injection	2	0	2	0
CVE-2009-4595	SQL injection	2	0	2	0
CVE-2009-1742	XSS	2	0	2	0
CVE-2009-4456	SQL injection	2	0	2	0
CVE-2009-3716	SQL injection	2	0	2	0

Table 6: Summary of system sanitizing effectiveness

CVE #	A: Vulnerable injection point amount (before protection)	B: Vulnerable injection point amount (after protection)	Protection effectiveness (A-B/A)
Experimental Web site	11	0	100%
CVE-2010-0122	5	0	100%
CVE-2009-4669	2	0	100%
CVE-2009-4595	2	0	100%
CVE-2010-1742	1	0	100%
CVE-2009-4456	1	0	100%
CVE-2009-3716	2	0	100%

The maximum amount of the false positives for a generic blacklist will equal to the amount of all special characters appearing in the Testing Bank 6 (i.e. greater than 18). The maximum amount of the false positives for a general whitelist will equal to the amount of all normal input strings including special characters (i.e. greater than 18). Table 5 shows that the maximum amount of the false positives for the sanitizing system equals to 2 while false negative is zero if the auto-tuning mechanism automatically adjusts the value of legalSpecialCharCount to 1. The results show that the sanitizing system can effectively lower flase positive. That is to say, the sanitizing system renders fewer false positives, compared to other systems using general whitelist or blacklist.

In order to show that the auto-tuning mechanism can automatically adjust constraining rule to effectively sanitizing new injection patterns, a looser constraining rule, such as Sanitizing_Sql_Rule 1, was used in meta-program at first. And then injection attack tools will launch injection attack to the vulnerable web applications listed in Table 2.

A looser constraining rule may induce more false negatives. Thus some malicious injection strings will pass sanitizer and are kept in Web access log. These injec-

tion strings will be translate to new injection patterns by the injection pattern generator and then sent to testing pattern table. Next the constraining rule generator will organize stricter constraining rule to lower false negatives.

Finally, we need to verify the effectiveness of the sanitizing agent for protecting against injection attacks. Table 6 presents the amount of vulnerable injection points before and after the protection of our system. For each vulnerable website, the amount of vulnerable injection points after protection is zero. The results prove that our sanitizer can effectively protect against injection attacks caused by improper input validation.

6 Conclusion

Vulnerable websites with the injection flaws are being attacked and damaged every day. Injection flaws are increasingly vulnerable and protecting them requires a system that can both ensure compliance today and meet the evolving needs of an organization for tomorrow. To meet the challenge, organizations should continue to be diligent by regularly performing vulnerability scanning and penetration testing. Unless a web application has a strong, centralized mechanism for validating all input from HTTP requests (and any other sources), injection flaws are very likely to exist. Therefore, organizations should select and deploy a system providing rapid protection to close the vulnerability gap, with minimal impact on operations. In this paper, we come up with a proposal that can be used as compensating controls to protect web applications while vulnerabilities exist and patching is not an immediate option. To improve the risk of injection flaws and reduce unnecessary limitation of input characters, we design an auto-tuning system to help validating input for each vulnerable injection point. The experimental results show that the system can effectively protect against injection attacks and lower false positives while compared with traditional methods.

Acknowledgments

This work was supported by the National Science Council of Taiwan under grants NSC 100-2218-E-412-001. I gratefully acknowledge the anonymous reviewers for their valuable comments.

References

[1] Antivirus Solutions, http://en.wikipedia.org/wiki/Antivirus_software, accessed on April 7, 2012.

[2] J. M. Chen, "An improved sanitizing mechanism based on heuristic constraining method," *Advanced Research on Electronic Commerce, Web Application, and Communication*, Communications in Computer and Information Science, vol. 144, pp.153-159, 2011.

[3] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, J. D. Lockwood, "Deep packet inspection using parallel bloom filters", in *Proceedings of the 11th Symposium for High Performance Interconnect*, pp.44-51, 2003.

[4] Ferruh, SQL Injection Cheat Sheet, http://ferruh.mavituna.com/sql-injection-cheatsheet-oku/, accessed on June 14, 2010.

[5] M. Fisk and G. Varghese, "Fast content-based packet handling for intrusion detection", *UCSD Technical Report CS2001-0670*, 2001.

[6] W. Halfond and A. Orso, "Combining static analysis and runtime monitoring to counter SQL injection attacks," in *Proceedings of the Third International ICSE Workshop on Dynamic Analysis (WODA'05)*, 2005.

[7] J. H. Hayes, A. J. Offutt, "Increased software reliability through input validation analysis and testing software reliability engineering", in *Proceedings of the 10th International Symposium on Software Reliability Engineering*, pp.199–209, 1999.

[8] Y. W. Huang, C. H. Tsa, T. P. Lin, S. K. Huang, D. T. Lee, S. Y. Kuo, "A testing framework for Web application security assessment," *Journal of Computer Networks*, vol.48, no.5, pp.739-761, 2005.

[9] IBM Rational Corp., Web Application Security Testing—App-Scan, http: //www-01.ibm.com/software/rational/offerings/websecurity/, accessed on Jan. 10, 2009.

[10] I. M. Kim, "Using Web application firewall to detect and block common web application attacks," *SAN Institute Technical Report*, 2011.

[11] J. C. Lin, J. M. Chen, "An automatic revised tool for anti-malicious injection," in *The Sixth IEEE International Conference on Computer and Information Technology*, pp.164, 2006.

[12] J. C. Lin, J. M. Chen and C. H. Liu, "An automatic mechanism for sanitizing malicious injection", in *Proceedings of the 9th International Conference for Young Computer Scientists*, pp.1470-1475, 2008.

[13] J. C. Lin, J. M. Chen, and H. K. Wong, "An automatic meta-revised mechanism for anti-malicious injection," in *Proceedings of Network-Based Information Systems*, LNCS 4658, pp.98-107, 2007.

[14] NVD (National Vulnerability Database), http: //nvd.nist.gov/, accessed on June 10, 2010.

[15] Open Source Web Application Firewall: ModSecurity, http://www.webresourcesdepot.com/open-source-we-application-firewall-modsecurity/, accessed on Jan. 15, 2009.

[16] OWASP, WebScarab Project, http: //www.owasp.org/webscarab/, accessed on Jan. 18, 2009.

[17] OWASP, http://www.owasp.org/index.php/Web_Application_Firewall, accessed on Jan. 11, 2010.

[18] OWASP, https://www.owasp.org/index.php/ Injection_Flaws, accessed on Jan. 11, 2011.

[19] J. C. Rabek, R. I. Khazan, S. M. Lewandowski, R. K. CunninghamRabek, "Detection of injected, dynamically generated, and obfuscated malicious code," *Defense Advanced Project Agency (DARPA)*, Copyright Association for Computing Machinery, ACM, 2003.

[20] RSnake, XSS (Cross Site Scripting) Cheat Sheet, http://ha.ckers.org/xss.html, accessed on Jun 12, 2010.

[21] A. Salem, "Intercepting filter approach to injection flaws," *Journal of Information Processing Systems*, vol. 6, no. 4, pp.563–574, 2010.

[22] Sanctum Inc., AppShield white paper, http://www.sanctuminc.com/, accessed on Jan. 11, 2009.

[23] SPI Dynamics, Web Application Security Assessment, SPI Dynamics Whitepaper, http://www.spidynamics.com/, accessed on Jan. 20, 2009.

[24] Z. Radwan, C. Gaspard, A. Kayssi, and A. Chehab, "Policy-driven and content-based Web services security gateway", *International Journal of Network Security*, vol. 8, no. 3, pp. 253–265, May 2009.

Jan-Min Chen received the Ph.D. degree in the Department of Computer Science and Engineer from Tatung University, Taiwan, in 2010. He is currently an assistant professor in the Department of Information Management at Yu Da University, Taiwan. His research interests include computer network security, computer network management and web application security. His email is ydjames@ydu.edu.tw.