2010

Detect and Sanitise Encoded Cross-Site Scripting and SQL Injection Attack Strings Using a Hash Map

Erwin Adi
BINUS University, Indonesia

Irene Salomo
BINUS University, Indonesia

DOI: 10.4225/75/57b66ecc3477a
Originally published in the Proceedings of the 8th Australian Information Security Mangement Conference, Edith Cowan University, Perth Western Australia, 30th November 2010

Detect and Sanitise Encoded Cross-Site Scripting and SQL Injection Attack Strings Using a Hash Map

Erwin Adi and Irene Salomo
School of Computer Science
BINUS International – BINUS University, Indonesia
eadi@binus.edu, irene.salomo@gmail.com

Abstract

Cross-Site Scripting (XSS) and SQL injection are the top vulnerabilities found in web applications. Attacks to these vulnerabilities could have been minimised through placing a good filter before the web application processes the malicious strings. However adversaries could craft variations on the attack strings in such a way that they do not get filtered. Checking through all of the possible attack strings was tedious and causes the web application performance to degrade. In this paper, we propose the use of a hash map as a data structure to address the issue. We implemented a proof-of-concept filter which we tested through an open-source web application to show that such filter could sanitise some attack strings that otherwise were too tedious to detect. Our evaluation included comparing the proposed solution with other existing ones such as prepared statements, input length limitation, white list and black list input validation; our proposed solution performed the most efficient.

Keywords
Cross-Site Scripting, XSS, SQL injection, hash map, attack strings, web application vulnerabilities.

INTRODUCTION

Cybercrime already cost businesses $1 trillion globally in 2008 alone (Mills, 2009). It included lost of intellectual property and expense to solve the problems. Hence, computer security cannot be considered as an insignificant issue anymore; people start to realise the urgency of protecting their computer system, particularly in business areas where every single failure in computer systems may result in financial loss worth hundreds to millions dollars, or bad reputation.

Cross site scripting, also known as XSS, was the most likely web vulnerability and was found in 7 out of 10 web applications (Grossman, 2007). The same source also showed that the Structured Query Language (SQL) injection attack was the top 5 vulnerability and was found in 1 out of 5 web applications. The increasing risks of attacks attributed to the above vulnerabilities were also prompted by available automated tools that enabled faster and much easier attacks. Even without much knowledge about database or web programming, attackers could simply exploit these vulnerabilities, while in the past attackers still used manual methods that required more effort and time.

There were some similarities on how to detect and prevent both XSS and SQL Injection vulnerabilities. The impact of the attack could have been minimised through good programming practices and use of filter, or sanitisation process. However, the existing solutions did not address how to detect encoded attack strings. Web browsers supported several encoding mechanisms i.e., HTML, URL, UTF-7 encodings, etc. HTML encoding would encode character < into *<* characters. As a result, when attacker tried to inject *<script>* as an XSS attack, input sanitisation would encode it into *<script>* so that XSS attacks would not be successfully launched. The problems with this approach were (Auger, Cross site scripting, 2009; Ollmann, n.d.; Stuttard & Pinto, 2008):

(i) Some sanitisations encoded or removed malicious strings in total. For instance a function removed an input string if it read *<script>* tag; an attacker could bypass this function using *<scr<script>ipt>* tags.
(ii) Since most HTML and JavaScript codes were written in lowercase, some filters only encoded lowercase characters. For instance, *<scRIPt>* might evade filters.
(iii) Some lower level programming treated null byte characters such as *\0* or *&00* as signs to end a sequence of strings or to stop processing. On the contrary, higher level programming languages could interpret them as regular inputs. This difference could let attackers escape sanitisation by injecting null byte character before a malicious script.
(iv) Input strings from the user's browser might be encoded differently to guarantee that varied data may be transmitted safely over HTTP. Canonicalization was the process of converting or decoding data into a common character set after that encoding process. If this canonicalization was completed after input sanitisation process, an attacker could inject his scripts successfully by encoding them firstly. Examples were *%3cscript%3e* or *%253cscript%253e* tags.

In this paper, we proposed to use a hash map to detect encoded XSS and SQL injection attack strings. We proved that it resulted in a more efficient way of detecting and sanitising the vulnerabilities through our experiment.

The remaining of this paper is arranged as follows. First, we discuss the existing solutions to detect and sanitise XSS or SQL injection (or both). Then we propose our design and explain how we implement the design. We show our testing in the *Results* section, which we then compare our results with the other solutions within the *Evaluation and Discussion* section.

EXISTING SOLUTIONS

Prepared statement was one of those features in many web programming languages to manage its SQL statement. Basically, application would obtain its SQL statement and execute it instantaneously. Prepared statement allowed developers to separate processes between input parsing and database logic. It was used to set up an input statement first, and then execute it many times with different parameters. SQL statements which were processed with this technique would be precompiled first before executed by the database management system (Fisk, n.d.; Kabutz, n.d.; Wiegenstein & Weidemann, 2007). Hence prepared statement was one of a method to prevent SQL Injection. Currently, popular web programming languages support this concept: Java with PreparedStatement class, and PHP with PHP Data Objects (PDO). Unfortunately, the improved performance happened only if that particular input statement was executed for several times. Otherwise there would be a round trip to the server for both input parsing and executing the database logic, and that made the technique cost more than the regular statement.

Generally, XSS attack strings required longer length of input string attacks than the normal input strings. Based on this assumption, some web developers limited the permitted input length. However, this approach was not very flexible and convenient for the users, especially when a longer input data (such as an address) was needed. Furthermore, some SQL injection attacks did not need long attack strings. A determined attacker might penetrate this defense by shortening the attack payload and removing unnecessary characters.

White list was a list of permitted or legal input strings in web applications. Any input that did not match any character on the list would not be accepted. The problem with this solution was the complexity in constructing a good white list (Presson, 2008; Wiegenstein & Weidemann, 2007). Web developers needed to exhaustively select what kind of characters he should allow in his application because there were a vast number of words in the world from all human languages. Furthermore, different type of input fields had different list of permitted inputs (e.g., phone numbers should allow only numeric characters, while city names permitted only alphabet characters). A possible result of having an inaccurate white list was to have a false positive alarm. It could be argued that a white list input validation did not deliver an efficient solution to prevent SQL injection or XSS attacks.

One alternative solution to address the above problems was through using input sanitisation. This was a process of transforming or sanitising original input strings into more secured strings before processing and outputting them. It was assumed that all input strings may be malicious and the application needed to anticipate them in advance. The transformation process usually involves encoding or decoding processes.

PROPOSED SOLUTION

We proposed to implement a hash map to detect the SQL injection and XSS attack strings. We tested our solution through web applications that was run in our local server; hence we did not attack real websites.

We deployed a simple web application from http://gotocode.com/ that provided free, open source, and database-connected web applications in various programming languages, like ASP, JSP, PHP, Perl, ColdFusion, and ASP.NET/C #. Because of our familiarity with the language, we decided to deploy a Java Server Pages (JSP) web application of *Employee Directory* as the attacked target to provide our proof of concept. The *Employee Directory* was a simple application for administrator to store and manage all company's department names. Some possible injection points (called sources), in which attacks could be launch to this web application were through the login form, inserting department form, and updating department form.

To provide a proof-of-concept for both SQL Injection and XSS attacks, we tried to inject malicious input strings into those sources. The following are some injected input strings:

a. `' OR 1=1--`
This input string tried to bypass authentication process for administrator because 1 = 1 is always true and -- symbols would comment all remaining SQL statements.

b. ' or 1=1#
It had a similar purpose with the previous script. However we encoded the first character by using HTML encoding and represented its ASCII code.

c. ' or 1=1#
This script also served the same purpose with the previous script, but the first character was encoded by using HTML encoding without representing its ASCII code.

d. <script>alert('xss')</script>
This script could be considered as the most basic cross site scripting attack, it would pop up an alert dialog box with "XSS" text.

e. <script>alert(document.cookie)</script>
This script would also pop up an alert dialog box with the value of current cookie element. However, we encoded some characters in this strings using HTML encoding scheme. For example, < was encoded into *<* and > was encoded into *>* in order to bypass certain input filtering.

f. <img src=script:A L
ERT('Q')/>
This script was encoded by using HTML encoding and represented by ASCII numbers. We injected this script in the inserting department form. A successful result would put an image icon in the list of the departments.

g. <table><tr><td>Lo</td><td><input type=text length=20>
This input string would generate an input text field in which other users could insert data in the form, as if it were a legitimate input field of the web application.

h. ?><ScRiPt>alert('xss')</ScRiPt>
This script would produce a similar result with script (c), but we tried to circumvent input filtering by initiating script with the '?' sign, and used some capital letters.

As the victim was implemented in Java, we understood that the language itself provided a HashMap object in its library. Hence we constructed a filtering code using this language. We would explain the implementation process in two parts: module implementation and integrated implementation.

In the *module implementation*, we constructed a stand-alone filter application that had not been integrated with the web application. This was to ensure that all logic and procedure in input filtering and sanitisation were accurately implemented, before it was run in conjunction with the web application. Constructing the most complete input filtering list was not the main objective of this study.

Hash table algorithm works by associating keys and their values in one-to-one mapping. There were two available objects that followed this algorithm in Java: the HashMap class and the Hashtable class. We decided to choose the HashMap class instead of the Hashtable class. HasMap was more efficient for us since it was not synchronized. Although the input filter we constructed might be accessed by multiple clients, it would be managed and modified by a single thread only. Therefore it did not need synchronized class. On the other hand, Hashtable was synchronized, would consume more resources and would compromise the performance.

Most of the defined malicious characters were stored into the HashMap as hash keys, while their encoded formats were defined as hash values.

During the *integrated implementation* phase, we integrated the input filtering module with the web application. Input validation and sanitisation was applied in common.jsp file since the all other functions called the sanitisation process code in the file. The sanitisation process modules were:

a. String toSQL(String value, int type)
This method would accept SQL input strings, including the malicious one.

b. boolean checkParam(javax.servlet.http.HttpServletRequest req, String tainted)

This method would accept HttpServletRequest's object that hold user request's input strings, and then stored that object in a tainted variable called param. In addition, it would call the hash map as explained in our module implementation. The hash map contained a list of malicious characters for the first input filtering process. This method would notify web application whether the tainted variable contained malicious characters or not.

c. boolean replaceParam(String untainted)
If the checkParam method discovered that the tainted variable contained malicious character and it needed to be reflected to the users, this method would be called first to sanitise all input strings inside the tainted variable. Therefore this method would produce untainted variable that would be reflected to the users.

We implemented our filtering mechanism within the above methods. The sanitisation process used the hash map as the look-up table. The following snippet of code showed the initializing process of the hash map as a look-up table:

```
HashMap hash_filter = new HashMap(100, 0.75f);
hash_filter.put("<", "&lt;");
hash_filter.put(">", "&gt;");
hash_filter.put("\"", "&quot;");
hash_filter.put("&", "&amp;");
hash_filter.put("&amp;", "&");
hash_filter.put("!", "&#33;");
hash_filter.put("\\", "&#92;");
hash_filter.put("#", "&#35;");
```

The sanitation logic worked through replacing the malicious character with its encoded format and appending it to the other safe characters. We named the web application version that implemented this filter as "secured", compared to its "unsecured" counterpart.

RESULTS

To test this input filtering and sanitisation functions, we injected some scripts and their encoded formats into the web applications and compared the result between the secured and the unsecured application. Table 1 shows the testing result.

Table 1. Comparison Testing Result from Unsecured and Secured Application

Injected Script	Sources	Sink	Result from Unsecured	Result from Secured
' or 1=1#	Login form	SQL function	Successful	Unsuccessful
' or 1=1#	Login form	SQL function	Unsuccessful	Unsuccessful
' or 1=1#	Login form	SQL function	Unsuccessful	Unsuccessful
<script>alert('xss')</script>	Inserting department form	Reflecting to users	Successful	Unsuccessful
<script>alert (document.cookie)</script>	Inserting department form	Reflecting to users	Successful	Unsuccessful
	Inserting department form	Reflecting to users	Successful	Unsuccessful
<table><tr><td>lo</td></tr><input type=text length=20>	Inserting department form	Reflecting to users	Successful	Unsuccessful
?><ScRiPt>alert('test')</ScRiPt>"	Inserting department form	Reflecting to users	Successful	Unsuccessful

The table shows that most scripts and their encoded format could be injected successfully to launch SQL Injection and Cross Site Scripting attacks towards the unsecured web application. The hash map input filtering and sanitisation functions could prevent both attack by filtering and sanitisation the malicious characters into legal input strings.

EVALUATION AND DISCUSSION

In this part, we compared the result with some other existing and possible solutions. Our experiment implemented HashMap as the data structure in the lookup table to filter all malicious characters and to sanitise them. Table 2 below shows HashMap was the most appropriate searching algorithm according to our study, compared with other algorithms, such as linear search, binary search trees, heaps, and brute-force search. It had a good running time which was $O(1)$.

Table 2. Search Algorithm Comparisons

Algorithm Name	Running Time	Usage	Evaluation
Linear search	$O(n)$	To search an *array* or *list* by checking items one at a time sequentially.	It is not very efficient to iterate all elements
Binary search trees	$O(h)$, where h is the height of tree	To search a binary tree where every node's left subtree has keys less than that node's key	All elements in tree must be sorted first, so it may take extra time and resources
Heaps	$O\log(n)$	To search a binary tree with child node that is always smaller than its parent node.	All elements in tree must be sorted first according to heap's properties, so it may take extra time and resources.
Brute-force search	$O(n^2)$	To systematically enumerate all possible candidates for the solution and checking whether each candidate satisfies the problem's statement.	It is simple to implement, yet its cost may grow quickly as the size of problem increases
HashMap	$O(1)$	To search a table that contains mapping between unique keys and their own values.	All elements are not necessary to be sorted and enumerated. Instead, a key or value is sufficient to locate an element. Besides, it can be easily implemented.

Comparison with Prepared Statement

Prepared statement was a common strategy to prevent SQL Injection. However, if the query statement would not be necessarily executed several times, this solution cost more than regular statement since there would be round trips to the database server, to process both input parsing and executing database logic. In addition, it could only prevent SQL Injection, and not Cross Site Scripting.

Unlike prepared statement, these HashMap–based input filtering and sanitisation functions could prevent both SQL Injection and Cross Site Scripting. It did not filter and sanitise input in the database server, thus there was no round trip to the database server.

Comparison with Attribute Length Limit of User Input Strings

Unlike attribute length limit, our input filtering and sanitisation functions did not limit the number of characters that the user inserted. No matter how long those input strings were, our design would filter and sanitise them. Therefore, this proposed solution was more flexible for the web application. Moreover, determined attackers normally had many strategies to shorten their injected input strings; hence good programming practices should not assume that limiting input length could bypass all malicious input strings.

Comparison with White List Input Validation

It was elaborative to construct a good white list to validate input because of the vast range of human languages. Hence we actually implemented a black list input validation solution by constructing a mapping between malicious characters and their encoded formats; we applied the black list technique as a sanitisation solution. The benefit of this approach was that certain input strings could still be displayed to the users after being sanitised, instead of rejecting them completely.

Comparison with Black List Input Validation

Black list input validation consisted of all malicious characters which should not be allowed in a web application. This solution was much easier and more flexible than white list input validation as programmers did not need to know all possibilities of user supplied input strings. Instead, they would construct a list of malicious input strings. However, attackers could bypass this validation and sanitisation solutions by many methods, including encoding and varying input strings (Auger, Improper input handling, 2010) (Wheeler, 2003).

Although our proposal also implemented a black list input validation solution, it attempted to anticipate the attackers' strategy. In our implementation, the HashMap's keys consisted of both malicious characters and their encoded formats that were mapped with their encoded formats and malicious characters as well. For instance:

```
hash_filter.put("&", "&amp;");
hash_filter.put("&amp;", "&");
```

As a comparison, other existing input filtering and sanitisation functions implemented linear search algorithm. As an example, this following snippet of code was a sanitisation function provided by GotoCode (http://gotocode.com/).

```
String toHTML(String value) {
    if ( value == null ) return "";
    value = replace(value, "&", "&amp;");
    value = replace(value, "<", "&lt;");
    value = replace(value, ">", "&gt;");
    value = replace(value, "\"", "&" + "quot;");
    return value;
  }
```

Therefore, our proposed solution could improve the existing black list input validation. Instead of iterating entire available filtering and sanitisation criteria, our filtering and sanitisation functions would locate certain entries by using their keys since hash map had mappings between keys and their values. No matter how many criteria a filtering had, it only needed to access this hash map table once by using the given key.

We encountered some obstacles during the development. First, we found that there were too many possible types of input strings to launch SQL Injection and XSS attacks. It was impossible to include all of them. However, the purpose of our study was to provide a proof-of-concept implementation of our design: to provide a simple and fast solution that addressed the problem of handling the variation of the attack strings.

Second, different web browsers might show different results towards the same security attacks. In the beginning we used three different web browsers: Internet Explorer 6.0, Mozilla Firefox 3.6.3, and Google Chrome 4.1. In fact each of them had different vulnerabilities and therefore each showed different reactions to the problem. This made us difficult to trace back if a problem was caused by the browser, or if it was caused by our ineffective solution. Therefore we finally focused more on Mozilla Firefox.

CONCLUSION AND RECOMMENDATIONS

In our proof-of concept experiment, we have shown that a HashMap was able to provide an easy to implement, a fast running time sanitisation function to detect encoded SQL injection and XSS attack strings. To implement our proposal into a more general practice, one needs to develop libraries or plug-ins for input filtering and sanitisation functions that can be used transparently by web developers. For instance, web developers could call a Java library or PHP built in function *htmlentities( )* without having to be aware that sanitisation has occurred.

Finally, there is an open possibility that the efficiency of the HashMap operation could be optimized as the attack factor grows. In Java, there are two factors that affect the efficiency of a hash algorithm: the initial capacity and the load factor. These are initialized during the call to the constructor. Because we have already shown that a hash map can be implemented in input filtering and sanitisation functions, future research can focus on how to optimize this algorithm.

REFERENCES

Auger, R. (2009, 12 30). *Cross site scripting*. Retrieved June 1, 2010, from The Web Application Security Consortium: http://projects.webappsec.org/Cross-Site+Scripting

Auger, R. (2010, January). *Improper input handling*. Retrieved June 1, 2010, from The Web Application Security Consortium: http://projects.webappsec.org/Improper-Input-Handling

Fisk, H. (n.d.). *Prepared statements*. Retrieved June 1, 2010, from MySQL developer zone: http://dev.mysql.com/tech-resources/articles/4.1/prepared-statements.html

Grossman, J. (2007, October). *WhiteHat website security statistics report.* Retrieved March 8, 2010, from WhiteHat Security: http://www.whitehatsec.com/home/assets/WPStatsreport_100107.pdf

Kabutz, H. M. (n.d.). *Closing database statements*. Retrieved June 1, 2010, from Javaspecialists.eu: http://www.javaspecialists.eu/archive/Issue116.html

Mills, E. (2009, January). *Study: Cybercrime cost firms $1 trillion globally*. Retrieved March 8, 2010, from Cnet News: http://news.cnet.com/8301-1009_3-10152246-83.html

Ollmann, G. (n.d.). *HTML code injection and Cross-site scripting: Understanding the cause and effect of CSS (XSS) vulnerabilities*. Retrieved June 1, 2010, from Technical Info: Making sense of security: http://www.technicalinfo.net/papers/CSS.html

Presson, M. (2008, May 2). *White list input validation, where it becomes hairy.* Retrieved June 2, 2010, from Coding insecurity: General guidance on how to make your applications more secure: http://coding-insecurity.blogspot.com/2008/05/whitelist-input-validation-where-it.html

Stuttard, D., & Pinto, M. (2008). *The web application hacker's handbook.* Indianapolis, IN: Wiley Publishing, Inc.
Wheeler, D. (2003, October). *Secure programmer: Validating input*. Retrieved June 10, 2010, from IBM Website: http://www.ibm.com/developerworks/linux/library/l-sp2.html

Wiegenstein, A., & Weidemann, F. (2007, November 2). *Input validation is no silver bullet against hacker attacks.* Retrieved June 1, 2010, from Virtualforge: We harden your software: http://www.virtualforge.de/whitepapers/input_validation.pdf