Client-Side Detection of SQL Injection Attack

Hossain Shahriar, Sarah North, and Wei-Chuen Chen

Department of Computer Science
Kennesaw State University
Georgia, 30144, USA
{hshahria,snorth}@kennesaw.edu, wchen10@students.kennesaw.edu

Abstract. Despite the development of many server-side approaches, SQL Injection (SQLI) vulnerabilities are still widely reported. A complementary approach is to detect the attack from the client-side (browser). This paper presents a client-side approach to detect SQLI attacks. The client-side accepts shadow SQL queries from the server-side and checks any deviation between shadow queries with dynamic queries generated with user supplied inputs. We propose four conditional entropy metrics to measure the deviation between the shadow query and dynamic query. We evaluate the approach with an open source PHP application. The results indicate that our approach can detect malicious inputs early at the client-side.

Keywords: SQL Injection, conditional entropy, client-side attack, web security.

1 Introduction

The starting point of SQL Injection (SQLI) [1] attack is the client-side (browser). If malicious inputs can be detected at the browse, then many SQLI attacks could be thwarted by preventing them to be supplied to the server-side. This could bring two benefits: (i) adding as an extra protection layer on top of existing server-side solutions (secure coding [3], code generation [2], dynamic analysis [6]), and (ii) working as complementary to scanner tools [4, 5] that do not have the knowledge of intended SQL query structure.

There are challenges for developing client-side SQLI attack detection approach. For example, the relevant form fields that require checking at the client-side must be informed by the server-side and priori information about the query structure needs to be known. The supplied information from the server-side should not contain sensitive information such as the original names of query tables and columns. The alteration of dynamic query structure needs to be checked at the client-side.

This paper attempts to address these issues. We propose a server-side assisted client-side SQLI attack detection framework. We extract query structures from the server-side and convert them to shadow queries. A shadow query is identical to an original query, except column names, table names, and input variables are replaced with arbitrary symbolic values. We measure the deviation of information content between shadow queries and actual queries (replacing symbolic values with supplied

X. Franch and P. Soffer (Eds.): CAiSE 2013 Workshops, LNBIP 148, pp. 512–517, 2013.

input values). We propose conditional entropy metrics (a form of measuring information content) as a basis to differentiate between benign and malicious queries at the client-side. If there is a deviation of the information content between a shadow query and a dynamic query (formed by replacing symbolic values with actual form field values), then an SQLI attack is detected at the client-side and a request is blocked. Otherwise, inputs supplied to a form are considered as benign and a request is allowed to the server-side. We evaluate our approach with an open source PHP web application containing known SQLI vulnerabilities. The results indicate that the approach can detect malicious attack inputs at the client-side.

The paper is organized as follows. Section 2 shows an example of SQLI attack followed by a brief discussion on the related works. In Section 3, the proposed approach is discussed in details. Section 4 discusses the experimental evaluation. Section 5 concludes the paper and discusses future work.

2 SQL Injection (SQLI) Attack and Client-Side Detection

We show a login form in Figure 1(a) with two input fields (*Login*, *Password*) whose values could contain malicious inputs for SQLI attacks. Figure 1(b) shows the HTML code of the form. When a user clicks on *Login* button, a request for accessing the website is performed with the supplied values. Here, the Login and Password inputs are accessed at the server-side through *$fLogin* and *$fPassword* variables. The *login.php* script is executed at the server-side which generates a dynamic SQL query to perform authentication based on the supplied values.

(a) Login form	(b) HTML of login form
Enter login and password Login admin' or 1=1 -- Password Login	<form name = “form1” action="login.php" method="post"> <input type=”text” name= “fLogin”> <input type= “text” name= “fPassword”> <input type= “submit” name= “Login”> </form>

Fig. 1. (a) A login form (b) HTML code of login form

```
1.  $login = $_POST[‘fLogin’];
2.  $pwd = $_POST[‘fPassword’];
3.  $qry = "select id, level from tlogin where uid =’”. $login. “’ and password =’”. $pwd. “’”;
4.  $result = mysql_query($qry);
5.  while($row = mysql_fetch_array($result)) { // authentication
6.      $_SESSION[’ID’] = $row['id'];
        ……..
7.  }
```

Fig. 2. PHP code for authentication

Figure 2 shows the PHP code snippet. Lines 1 and 2 extract from *fLogin* and *fPassword* fields of a request into *$login* and *$pwd* variables, respectively. The inputs are not filtered and concatenated with other strings for generating the dynamic SQL

query at Line 3. Line 4 executes the query and Line 5 performs the authentication based on the presence of at least one row in the result set. If the provided credential information is matched, the current session id is set with the *id* field from the table.

If an attacker supplies malicious inputs as shown in Figure 1(a), the query becomes *select id, level from tlogin where uid ='' or 1=1 --' and password ='*'. The resultant query is a tautology (the query after the comment symbol "--" is ignored). The remaining condition *uid='' or 1=1* is evaluated as *true*. Therefore, the malicious input alters the intended query structure implemented by a programmer.

(a) Revised HTML form

```
<form name = "form1" action="login.php" method="post">
  <input type="text" name= "fLogin">
  <input type= "text" name= "fPassword">
  <input type= "hidden" name= "sQuery"
    value= "select c1, c2 from t1 where c3 =v1 l1 c4 = v2">
  <input type= "hidden" name= "chkField" value ="fLogin,fPassword">
  <input type= "hidden" name= "substitue" value ="v1,v2">
   <input type= "submit" name= "Login" onclick= "chkSQLI ()">
</form>
```

(b) JavaScript code for checking SQLI attack

```
1.  function chkSQLI () {
2.    var sqry = document.form1.sQuery;
3.    var entropySqry = condEntropy(sqry);
4.    var aqry = subs (sqry, document.form1.chkField, document.form1.substitue);
5.    var entropyAqry = condEntropy(aqry);
6.    if (entropySqry == entropyAqry)
7.      document.form1.submit();
8.    else
9.    alert ("SQL injection attack inputs are present in supplied values");
10. }
```

Fig. 3. (a) Modified HTML form (b) JavaScript code for checking SQLI attack

We now show how our approach can detect the attack from the client-side. We modify the HTML form as shown in Figure 3 (a) by adding three hidden fields. They represent server-side provided information of the shadow query (*sQuery*), the form fields that need to be checked (*chkField*) for malicious inputs, and the values in shadow query that need to be substituted with the form field values (*substitute*). A JavaScript code is supplied from the server-side to check SQLI attack inputs at the client-side. The method *chkSQLI* is shown in Figure 3(b). It extracts the shadow query in Line 2. Line 3 computes the conditional entropy of the shadow query (*entropySqry*). Line 4 substitutes shadow query's symbolic values (specified at *document.form1.substitue* as comma separated field values) with form fields (specified at *document.form1.chkField* as comma separated values). The conditional entropy (*entropyAqry*) of the actual query is obtained at Line 5. Line 6 compares the value of *entropyAqry* and *entropySqry*. If there is a match, the inputs are considered as benign and the form is submitted to the remote website (Lines 6-7). If there is a mismatch, a warning message is generated for SQLI attack related inputs (Line 9).

We now discuss about the framework briefly, server-side shadow query generation process, client-side conditional entropy computation technique in Section 3.

3 Client-Side SQLI Attack Detection Approach

First, server-side script is analyzed to identify HMTL forms that contain input fields. Then the SQL queries present in the script code is extracted and we find the relevant set of input fields in forms that contribute values during dynamic query generation process. The SQL query is simplified to generate a shadow query. This is done by replacing table, column, logical operator, and variable values with symbolic values. This step prevents revealing sensitive information at the client-side. The HTML forms are modified by adding hidden fields with the following: (i) the shadow query, (ii) fields relevant to dynamic queries and whose values need to be replaced with appropriate symbols in the shadow query, and (iii) the relevant symbols from shadow query that need to be replaced with user provided inputs.

The modified script is deployed in a web server. From a browser, when a user requests a web page to the server, the server-side returns the modified pages containing the revised HTML forms. When a user resubmits a form after providing inputs in form fields, the client-side checking takes place (JavaScript code). The checking is done by computing the conditional entropy of the shadow query and the actual query by substituting symbols with user provided input values. If any deviation is found, the inputs are flagged as malicious. The request is not forwarded to the server-side. Otherwise, the inputs are considered as benign and the request is forwarded to the server-side. We now briefly discuss about conditional entropy and its application for detecting client-side SQLI attacks.

Conditional entropy is a measurement technique from the area of information theory [7]. Given that we have a set of random variables, and we know the outcome of one variable in advance, the randomness of the remaining variables can be computed with the conditional entropy metric. For a given SQL query, if we have the priori information about the query type (select, insert), we can measure the information content about certain parts of the query. This allows us to reduce the computation time.

Let us assume that q be a query present in a program, $T=\{t_1, t_2, \ldots, t_M\}$ be the set of tables being used in the query, $C=\{c_1, c_2, \ldots, c_L\}$ be the set of all columns that can be selected, inserted, deleted, or updated in q, $V=\{v_1, v_2, \ldots, v_P\}$ are the set of values that are used in where conditions or setting values, $L=\{l_1, l_2, \ldots, l_P\}$ are the logical operators present in q, and $O=\{o_1, o_2, o_3, o_4\}$ which represents the *select* (o_1), *insert* (o_2), *update* (o_3), and *delete* (o_4) operation that can be performed in the query q.

We define four conditional probability functions as follows:

$P(c|o)$ is the conditional probability of column's presence given that we know the operation type of q, where $o \varepsilon O$.

$P(v|o)$ is the conditional probability of value v being used in a query's where condition or set values given that we know the operation type of q.

$P(t|o)$ is the conditional probability of table t being accessed or modified in a query given that we know the operation type of q.

$P(l|o)$ is the conditional probability of performing logical operation (in where condition) in q, given that we know the operation type of q.

We now define four conditional entropies of a query *q* as follows.

Given that we know the operation type of a query, the conditional entropy of column (denoted as H_c) can be computed as follows (*Equation (i)*):
$H_c(c|o) = -\sum_{c \varepsilon C} P(c,o)\ log\ P(c|o)$... *(i)*

H_c allows us to detect SQLI attacks where altered query selects additional columns (*e.g.*, *UNION select (1, 1)*) or reduces columns.

Given that we know the operation type of a query, the conditional entropy of values (denoted as H_v) can be computed as follows (*Equation (ii)*):
$H_v(v|o) = -\sum_{v \varepsilon V} P(v,o)\ log\ P(v|o)$... *(ii)*

H_v allows us to detect attacks where altered query may set new values to fields not intended by the original query (*e.g.*, tautology has 1=1 in attack signatures where 1 is assumed as a value).

Given that we know the operation type of a query, the conditional entropy of table (denoted as H_t) can be computed as follows (*Equation (iii)*):
$H_t(t|o) = -\sum_{t \varepsilon T} P(t,o)\ log\ P(t|o)$... *(iii)*

H_t allows us to detect attacks where altered query may perform additional operations on table not intended by the original query (*e.g.*, piggybacked queries selecting arbitrary tables).

Given that we know the operation type of a query, the conditional entropy of logical operation (denoted as H_l) can be computed as follows (*Equation (iv)*):
$H_l(l|o) = -\sum_{l \varepsilon L} P(l,o)\ log\ P(l|o)$... *(iv)*

H_l allows us to detect attacks where altered query may perform additional logical operations on table not intended by the original query (*e.g.*, tautology) as well as reduce the number of operators.

4 Evaluation

We evaluate the proposed approach using an open source PHP application named *PHP-Address book* (address and contact manger). We analyze 230 source files that have 38 forms, 157 input fields, 43 select, 11 update, 7 insert, and 7 delete queries.

We instrument the application with JavaScript code. We deploy the application in an Apache web server with MySQL server acting as the database. We then visit the application so that we can obtain the modified HTML forms. We supply malicious inputs (tautology, union, piggybacking) in each of the form fields. The approach detects the supplied malicious inputs in form fields.

5 Conclusions

This paper develops a client-side SQLI attack detection framework. Our approach relies on the generation of shadow queries as well as encoding information in HTML

forms to enable the necessary checking at the client-side. We measure the deviation between expected query and altered query with four conditional entropy metrics. The approach detects malicious inputs causing SQLI at the server-side early and relieves the server-side for additional checking on SQLI attacks. We evaluate the proposed approach with an open source PHP application and our approach detects the presence of malicious inputs early at the client-side before being passed to the server-side.

Our future plan includes extending the approach for stored procedures, and evaluating our approach with more web applications. Moreover, we are planning to extend the concept for detecting other relevant attacks such as cross-site scripting. Currently, our approach focuses only SQLI attacks caused by providing malicious inputs in forms. So, we will work on identifying non-form input-based SQLI attack detection. Further, our goal also includes comparing the performance of client-side detection approach with other scanner tools.

Acknowledgement. This work was funded in part by Kennesaw State University College of Science and Mathematics Faculty Summer Research Award Program.

References

1. The Open Web Application Security Project (OWASP), https://www.owasp.org/index.php/Top_10_2010-Main
2. Johns, M., Beyerlein, C., Giesecke, R., Posegga, J.: Secure Code Generation for Web Applications. In: Massacci, F., Wallach, D., Zannone, N. (eds.) ESSoS 2010. LNCS, vol. 5965, pp. 96–113. Springer, Heidelberg (2010)
3. Antunes, N., Vieira, M.: Defending Against Web Application Vulnerabilities. IEEE Computer 45(2), 66–72 (2012)
4. W3af, Open Source Web Application Security Scanner, http://w3af.org
5. SQL-inject-me, https://addons.mozilla.org/en-us/firefox/addon/sql-inject-me
6. Agosta, G., Barenghi, A., Parata, A., Pelosi, G.: Automated Security Analysis of Dynamic Web Applications through Symbolic Code Execution. In: Proc. of the 9th International Conference on Information Technology: New Generations (ITNG), Las Vegas, NV, pp. 189–194 (April 2012)
7. Cover, T., Thomas, J.: Elements of Information Theory. John Wiley and Sons (2006)