Multi-hashing for Protecting Web Applications from SQL Injection Attacks

Yogesh Bansal, Jin H. Park*
Computer Science, California State University, Fresno, CA 93740, U.S.A.

* Corresponding author. Email: jpark@csufresno.edu
Manuscript submitted July 10, 2014; accepted January 24, 2015.
doi: 10.17706/ijcce.2015.4.3.187-195

Abstract: SQL injection is a type of frequently reported security attacks on database-driven web applications in which attackers execute unauthorized query operations to access information. In this paper, we describe the design and implementation of an efficient protection scheme against the SQL injection attacks based on a multiple-hashing mechanism. The proposed protection system model consists of three phases, which are registration, login and validation phases, and database is divided into product and query databases. By using multiple hashing operations the proposed scheme achieves higher efficiency than conventional schemes, which do not use sophisticated hashing operations. The scheme is implemented with HTML, PHP and MySQL, and cryptographic hashing function SHA-512 is used in the coding. Our experimental results show that the proposed scheme achieves very high level of security gain with negligible amount of time overheads compared to the conventional methods

Key words: Authentication, hashing, SQL injection attack, web application.

1. Introduction

Web applications are ever demanding software used in modern days computing devices, which are connected to the Internet, and they provide a wide variety of services to various organizations and individual users. A typical web application receives users' requests from the browser, interacts with the back-end database and returns relevant information to the users. The back-end database often contains sensitive user data and thus, it attracts malicious users, i.e., attackers. SQL injection is one of the techniques used by attackers to extract or destroy important users' data. There might exist a certain level of vulnerabilities in web applications and that allows attackers to inject harmful SQL query segments during user input session and obtain unauthorized accesses to database. An unauthorized user can read or modify existing data, make the data unavailable to other users, or even corrupt the database server. Some well-known types of SQL injection attacks (SQLIAs) are: 1) tautology attacks in which a conditional statement is inserted in the query to make the condition always true; 2) union attacks in which keyword UNION is used in the query to perform illegal operations on the database; 3) logically incorrect query attacks, which might identify types of data or gather overall information about database or tables; and 4) piggy back attacks in which an injected query is added to the original query.

According to the OWASP report [1], SQL injection attacks are the top most threat to web applications. Since last decade there have appeared various detection and/or prevention methodologies addressing the SQL injection attacks in the literature [2]-[12]. However, each approach has it's own limitations on the scope and efficiency of addressing SQLIAs, and no single approach provides a comprehensive solution to various SQLIAs.

In this paper, we present the design and implementation of an efficient protection scheme against SQL injection attacks. The proposed scheme is based on a two-level hashing mechanism, which provides high protection rate with affordable time overheads. To support the scheme a separate database, query database, is used in addition to the product database to store and access the second-level hash code and authentication information during registration and login phases, respectively.

The reset of this paper is organized as follows. In Section 2, some dominant recent techniques of handling SQL injection attacks are briefly reviewed. In Section 3, the concept, system model and rationale of the proposed protection scheme are described. In Section 4, implementation details and experimental results are provided and finally, Section 5 concludes the paper.

2. Related Work

In this section, we briefly review some recent approaches of detecting and/or preventing SQL injection attacks on web applications.

An analysis method based on Honeynet and Honeypot technologies is described in [2]. In this work, the SQL injection detection scheme is developed based on the detection principles and multiple operating systems run on different databases simultaneously. Another detection methodology based on the profiling idea is described in [3]. The technique used in this approach is that healthy database behaviors are extracted and encoded in a XML profile and a data mining technique with finger printing is used to identify malicious queries. A relatively simple detection scheme used in [4] is based on both static and dynamic analysis methods. In this scheme, parameters are separated from query and a generalized algorithm based on static and dynamic analysis is used to detect whether the parameters are genuine or infected. In the prediction scheme used in [5], input sanitization routines are classified and static code attributes are used to predict SQL injections based on the types of the routines. An automatic transformation scheme, which converts a vulnerable web application code to a safe code is described in [6]. In this scheme, user intended queries are constructed by the method of dynamically running an application with candidate inputs. An approach proposed in [7] searches for vulnerabilities, including SQL injection, in web applications based on the network recording. In this approach, network forensic techniques and tools are used to analyze network packets containing get and post requests of a web application.

There have also appeared a certain number of approaches of enforcing authentication/encryption mechanisms to prevent SQL injections on web applications. A prevention scheme used in [8] uses a randomization based encryption technique. In this scheme, each character in the input value is substituted with one of four random values stored in the lookup table to decrease the probability of decrypting those values by hackers. An authentication scheme proposed in [9] uses advance encryption standard (AES) and encrypted user name and password are used to set a unique secret key for each user or client. A hashing method proposed in [10] uses user name and password to make a hashing value, which is created when a user account is created.

Of course, there have appeared a variety of approaches addressing SQLIAs in the literature, which are not mentioned in this section. A couple of recent survey works analyzing detailed features of some dominant approaches are found in [11], [12].

3. Proposed Scheme

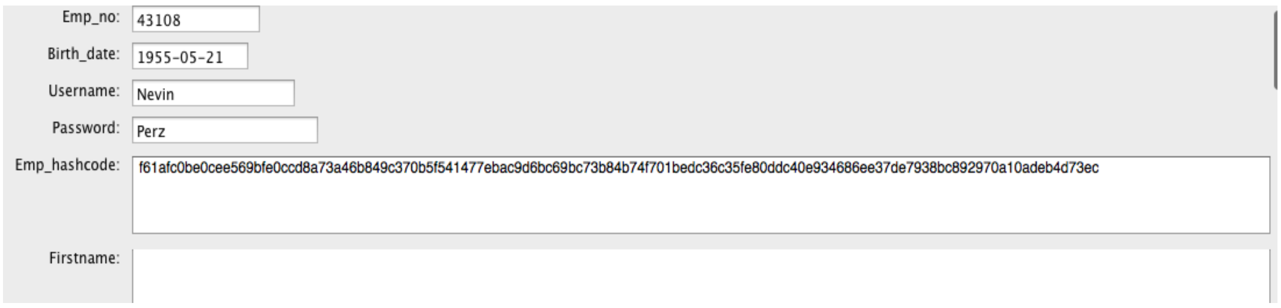
In this research, we enforce the authentication mechanism of web applications with multi-level hashing, which uses first and second hash codes, and multiple databases (product and query databases). To generate hash codes we utilize an existing encryption API, SHA-512 [13], [14]. We also use regular expression search and replacement APIs to remove html tags from user inputs. Our proposed system model consists of three phases, which are registration phase, login phase and validation phase.

3.1. Description of System Model

3.1.1. Registration phase

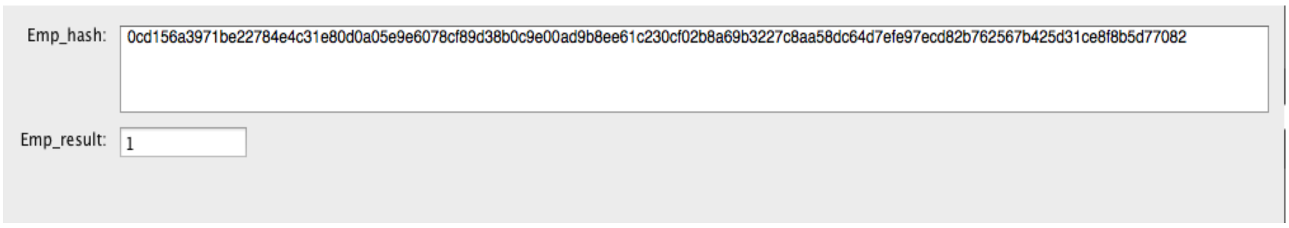
In the first phase of the model, a new user needs to register with providing a unique combination of a user name and a password through a web portal user interface. Upon submitting the user's credentials, data are sent to the server. At the server site, input validation operations, which include regular expression search and replacement, are performed and a hash code is generated from the SHA-512 encryption algorithm based on the combination of username and password provided by the user. This unique hash code, i.e., first hash code, is stored in the product database along with user's other data including address, email, phone number, etc.

During the registration process, a database query to be used in the login phase is predetermined and used to create another hash code, i.e., second hash code. For example, "select username, password from user where userhashcode=$Hashcode;" is a predetermined query to extract the detailed information of a user from database during the login phase. The value of the first hash code, i.e., "userhashcode" in the query, is generated from the username and password combination provided by the user, and the second hash code is then generated from applying the same hashing operation, SHA-512, on the predetermined login query with the first hash code. The second hash code and the predetermined login query information are stored in the query database, which is separated from the product database for the sake of modular design. Thus, in our system model, two-level hash codes are generated by identical hashing algorithms and stored in two different databases, i.e., product database for the first hash code and query database for the second hash code. Fig. 1 shows sample first and second hash codes (for a registered user) stored in the product and the query databases, respectively.



Emp_no:	43108
Birth_date:	1955-05-21
Username:	Nevin
Password:	Perz
Emp_hashcode:	f61afc0be0cee569bfe0ccd8a73a46b849c370b5f541477ebac9d6bc69bc73b84b74f701bedc36c35fe80ddc40e934686ee37de7938bc892970a10adeb4d73ec
Firstname:	

(a). First hash code in the product database.



Emp_hash:	0cd156a3971be22784e4c31e80d0a05e9e6078cf89d38b0c9e00ad9b8ee61c230cf02b8a69b3227c8aa58dc64d7efe97ecd82b762567b425d31ce8f8b5d77082
Emp_result:	1

(b). Second hash code in the query database.

Fig. 1. Sample hash codes.

The last operation in the registration phase is that the server sends back a confirmation message to the client (user interface). Fig. 2 illustrates the processing done in the registration phase.

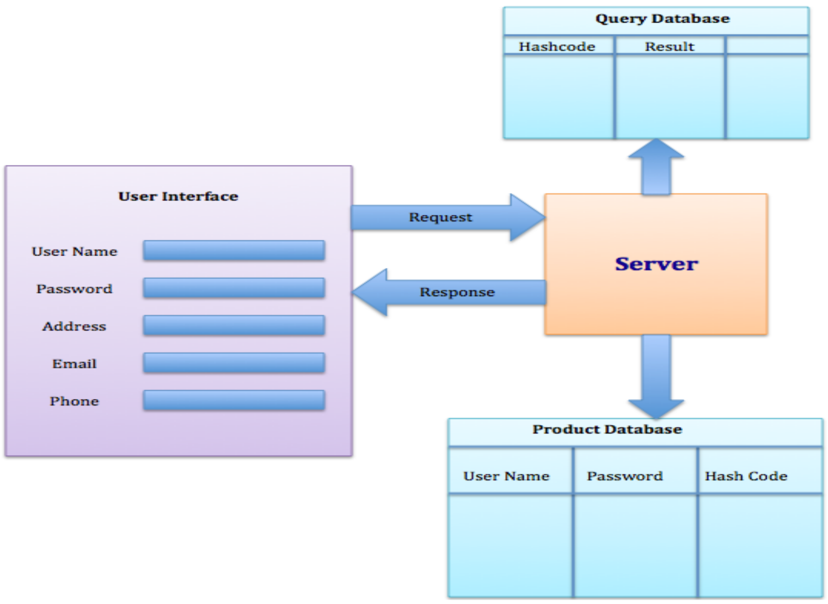


Fig. 2. Registration phase.

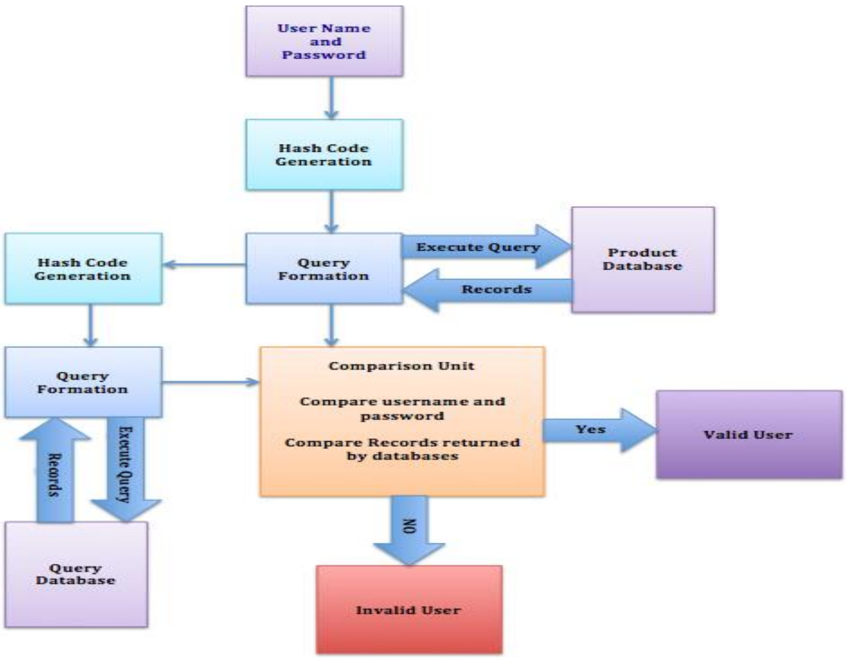


Fig. 3. Login and validation phases.

3.1.2. Login phase

When a registered user logs in username and password are passed to the server and the server generates a hash code, i.e., first hash code, by applying the SHA-512 encryption algorithm as it is done in the registration phase. This hash code is used to get the user information (records) from the product database, as well as used with the login SQL query to generate the second hash code via the SHA-512 encryption algorithm again. The security gain from using the first hash code to access the product database, instead of using the user name and password directly, is that it prevents SQL injections during the login process. The second hash code will be used in the validation phase to enforce the security of the web application against SQLIAs. In more detail, the second hash code will be intruded in the SQL query (e.g., select) to retrieve the predetermined result, i.e., the number of records in the product database for the user, from the query database. A sample query is shown below.

SELECT × FROM employees WHERE emp_hashcode =
'0cd156a3971be22784e4c31e80d0a05e9e6078cf89d38b0c9e00ad9b8ee61c230cf02b8a69b3227c8aa58d
c64d7efe97ecd82b762567b425d31ce8f8b5d77082'

3.1.3. Validation phase

Validation phase is tightly attached to the login phase to protect the web application from SQLIAs. In the validation phase the two hash codes, which are generated in the login phase, are used. When the first hash code is available in the login phase the user's records are extracted from the product database via a predetermined login query. Since the first hash code is a unique value, it always retrieves a single user's data (records). The retrieved authentication information, i.e., user name and password, are compared to the login user name and password for identity. As described earlier, it is the first level protection to use the first hash code as the key to extract the user's credentials from the product database.

Note that in the registration phase, the expected result of each predetermined login query for each registered user is stored in the query database. The second hash code, which is formed in the login phase, is also used in the validation phase as the key to retrieve the expected result stored in the query database. Then, the number of records retrieved from the product database via login query (with the first hash code as the key) and the expected number of records retrieved from the query database (using the second hash code as the key) are compared for validation again. If they are identical, the validation is positive, otherwise the validation is determined as unauthorized access. For example, if an attacker uses a SQL injection in the login phase to pull out information illegally, the number of records from the extracted data and the number of records stored in the query database do not match and the information is protected. Fig. 3 illustrates the processing in the login and the validation phases, and the rationale behind the proposed two-level hashing mechanism is provided in the next subsection.

3.2. Rationale behind the Proposed Two-Level Hashing

Although level-1 hashing provides pretty high-level protection against SQLIAs it cannot handle all the cases. Suppose that during the registration phase a hacker registers with username ' '*OR* 1=1- - and password *'Password'*. Then the protection system with only level-1 hashing would fail during the login phase. In the registration phase the system accesses the username and the password, and generates a hash code based on them, i.e.,

95493b06e341122b00506e1bdad62f53d62de0c96009bad8092a41d7feaa513f6df1e34326262609a28055
20e9679ce163159da98b1bc24c274cb7d4cc535dda

At the login time, when a user enters username ' '*OR* 1=1 - - and password *'Password'* the system generates the same hash code and the login query is generated based on them, i.e.,

SELECT × FROM employees WHERE emp_hashcode =
'95493b06e341122b00506e1bdad62f53d62de0c96009bad8092a41d7feaa513f6df1e34326262609a2805
520e9679ce163159da98b1bc24c274cb7d4cc535dda' AND username = ' '*OR* 1=1-- AND password = '
Password '

When the system uses these username, password and hash code to authenticate the user it would always return all the records stored in the database because the query would always be true.

The problem described above can be resolved with the two-level hashing mechanism used in our approach. In our current experiment, we store the second hash code and the predetermined result (number of records for each registered user) of the login query into the query database during the registration phase (refer Fig. 1(b)). In the case that the login query returns incorrect number of records, which is different from the predetermined result accessed from the query database, the activity is considered as an SQLIA.

4. Implementation and Experimental Results

4.1. Implementation

The proposed protection system model is implemented with HTML, PHP and MYSQL in the MacOSX environment. Net Beans IDE 7.2 and MAMP Server are used as the web server, and predefined PHP commands, such as "preg_match‟ and "stripslashes" are also used in our practice. As described in the previous section, we implemented database with separate modules, i.e., product database for storing user credentials and query database for storing query information. This separate database manipulation makes the proposed protection system scalable and portable.

We summarize the overall logic used in our implementation as follows:

Step 1. Access username and password during registration;

Step 2. Generate the first hash code based on the combination of username and password;

Step 3. Store user credentials with the first hash code in the product database;

Step 4. Generate the second hash code based on the first hash code and the predetermined login query;

Step 5. Store the second hash code in the query database together with the login query predetermined result (number of records in the product database);

Step 6. In the login phase, access username/password and generate the first hash code based on them;

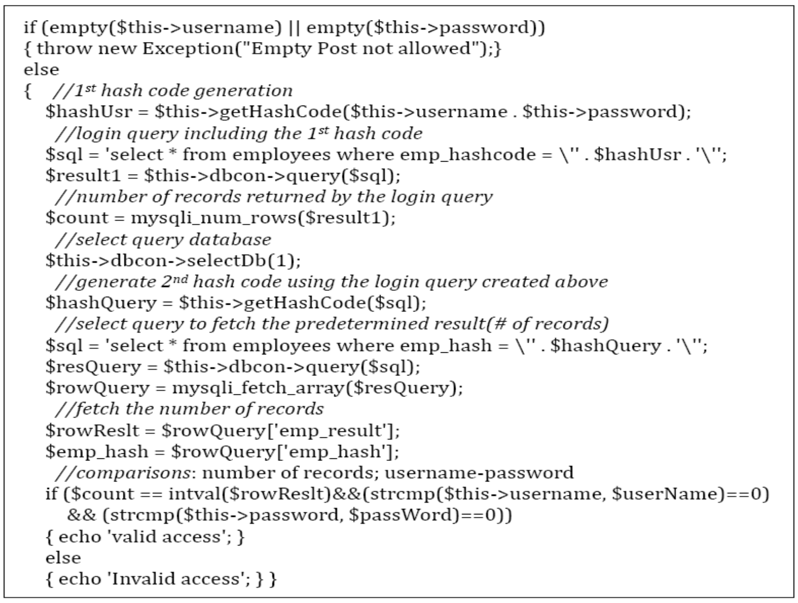
Step 7. Intrude the first hash code in the login query and retrieve the user's data from the product database;

Step 8. If the user's credentials are found, check the login username and password with the ones from the product database for validation; If they do not match, return error and terminate;

Step 9. Generate the second hash code from the first hash code and the login query;

Step 10. Use the second hash code as the key to get the record (predetermined result) from the query database;

Step 11. Compare the predetermined result stored in the query database (from step 10) to the number of records returned from Step 7 for validation. If they do not match, return error.

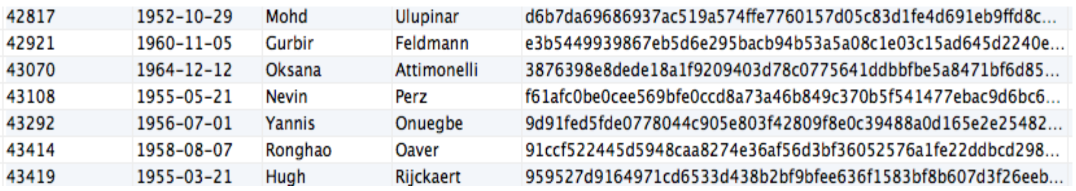


```
if (empty($this->username) || empty($this->password))
{ throw new Exception("Empty Post not allowed");}
else
{   //1st hash code generation
   $hashUsr = $this->getHashCode($this->username . $this->password);
     //login query including the 1st hash code
   $sql = 'select * from employees where emp_hashcode = \'' . $hashUsr . '\'';
   $result1 = $this->dbcon->query($sql);
     //number of records returned by the login query
   $count = mysqli_num_rows($result1);
     //select query database
   $this->dbcon->selectDb(1);
     //generate 2nd hash code using the login query created above
   $hashQuery = $this->getHashCode($sql);
     //select query to fetch the predetermined result(# of records)
   $sql = 'select * from employees where emp_hash = \'' . $hashQuery . '\'';
   $resQuery = $this->dbcon->query($sql);
   $rowQuery = mysqli_fetch_array($resQuery);
     //fetch the number of records
   $rowReslt = $rowQuery['emp_result'];
   $emp_hash = $rowQuery['emp_hash'];
     //comparisons: number of records; username-password
   if ($count == intval($rowReslt)&&(strcmp($this->username, $userName)==0)
      && (strcmp($this->password, $passWord)==0))
   { echo 'valid access'; }
   else
   { echo 'Invalid access'; } }
```

Fig. 4. Code segment for validation.

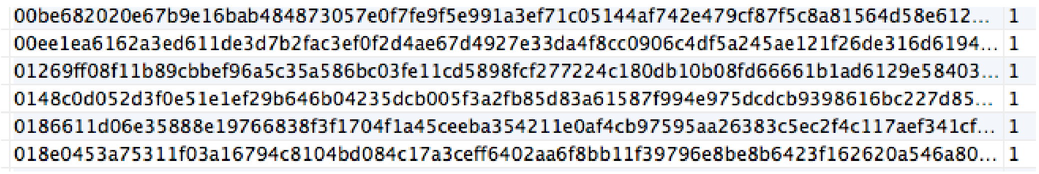
Steps 1-5 are implemented in the registration phase and illustrated in Fig. 2, and Steps 6-11 are implemented in the login and the validation phases and illustrated in Fig. 3. The code segment shown in Fig. 4 is used in the login and the validation phases to protect a web application from SQLIAs.

A sample product database table, which stores registered users' data and first-level hash codes, and a sample query database table, which stores second-level hash codes and predetermined results (i.e., the number of records in the product database for each user), are shown in Fig. 5.



42817	1952-10-29	Mohd	Ulupinar	d6b7da69686937ac519a574ffe7760157d05c83d1fe4d691eb9ffd8c...
42921	1960-11-05	Gurbir	Feldmann	e3b5449939867eb5d6e295bacb94b53a5a08c1e03c15ad645d2240e...
43070	1964-12-12	Oksana	Attimonelli	3876398e8dede18a1f9209403d78c0775641ddbbfbe5a8471bf6d85...
43108	1955-05-21	Nevin	Perz	f61afc0be0cee569bfe0ccd8a73a46b849c370b5f541477ebac9d6bc6...
43292	1956-07-01	Yannis	Onuegbe	9d91fed5fde0778044c905e803f42809f8e0c39488a0d165e2e25482...
43414	1958-08-07	Ronghao	Oaver	91ccf522445d5948caa8274e36af56d3bf36052576a1fe22ddbcd298...
43419	1955-03-21	Hugh	Rijckaert	959527d9164971cd6533d438b2bf9bfee636f1583bf8b607d3f26eeb...

(a). Sample product database table.



00be682020e67b9e16bab484873057e0f7fe9f5e991a3ef71c05144af742e479cf87f5c8a81564d58e612...	1
00ee1ea6162a3ed611de3d7b2fac3ef0f2d4ae67d4927e33da4f8cc0906c4df5a245ae121f26de316d6194...	1
01269ff08f11b89cbbef96a5c35a586bc03fe11cd5898fcf277224c180db10b08fd66661b1ad6129e58403...	1
0148c0d052d3f0e51e1ef29b646b04235dcb005f3a2fb85d83a61587f994e975dcdcb9398616bc227d85...	1
0186611d06e35888e19766838f3f1704f1a45ceeba354211e0af4cb97595aa26383c5ec2f4c117aef341cf...	1
018e0453a75311f03a16794c8104bd084c17a3ceff6402aa6f8bb11f39796e8be8b6423f162620a546a80...	1

(b). Sample query database table.

Fig. 5. Sample product and query database tables.

4.2. Experimental Results

To test the performance of the proposed protection approach a group of 20 individuals have been asked to try SQL injection attempts to our web application software, which is an experimental software manipulating 1200 employees' records. Each person is given 10 chances to perform attacks to retrieve unauthorized information and we computed the ratio as the number of successful hackers, who retrieved unauthorized information at least once, based on the total number of individuals, i.e., 20. We tested with different numbers of records in the product database to observe the effect of the database size on the protection rate and the execution time. For the comparison purpose in our practice, we also implemented the schemes with 1-level hashing and with no hashing.

The graph in Fig. 6 shows average execution time differences among those three schemes, i.e., no hashing, 1-level hashing and 2-level hashing. As shown in the graph, execution time increases somewhat linearly according to the increasing size of the product database, but there is no significant difference among the tested schemes, except the last case with record size 1275.

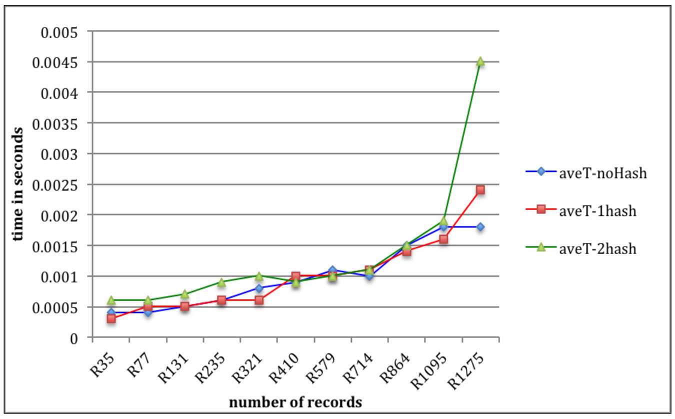


Fig. 6. Execution time measurement.

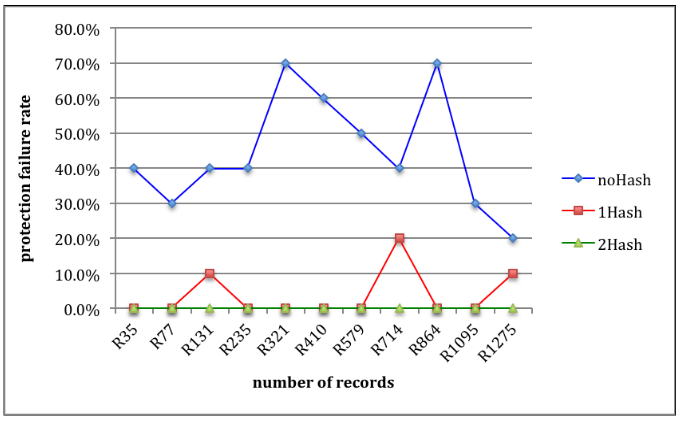


Fig. 7. Protection performance measurement.

The graph in Fig. 7 shows the protection failure rates of the three schemes. As shown in the graph, the proposed scheme (2Hash) could protect the web application from SQLIAs perfectly (0% failure rate for all the cases tested), and it is observed that the protection performance is independent from the database size.

5. Conclusion and Discussion

In this paper, an efficient 2-level hashing approach of protecting web applications from SQLIAs is described. The proposed protection system model consists of registration, login and validation phases and is implemented in the Mac OSX environment with HML, PHP and SQL. In our scheme, user name/password and login query are encrypted by SHA-512 hashing function to form the first and the second level hash codes, respectively. By using the 2-level hashing mechanism with separate product and query databases, we gained very high-level protection rate. In fact, in our practice the system never failed though the practice is limited with 20 people and 10 trials each. Based on our experimental results, we conclude that the proposed protection scheme is highly effective for preventing SQL injection attacks with negligible amount of time overheads.

The proposed protection scheme is portable and we plan to test the scheme further with diversified web applications and diversified group of people in the future work.

References

[1] OWASP Top Ten Project (2013). Retrieved March 10, 2014, from https://www.owasp.org/index.php/category:owasp top ten project

[2] Yin, Z., Niu, Z., & Tong, F. (2013). The design of SQL injection analysis system based on Honeynet. *Proceedings of the Int'l Multi Conference of Engineers and Computer Scientists* (pp. 403-406). Hong Kong.

[3] Salama, S. E., Marie, M. I., El-Fangary, L. M., & Helmy, Y. K. (2012). Web anomaly misuse intrusion detection framework for SQL injection detection. *International Journal of Advanced Computer Science and Applications, 3(3),* 123-129.

[4] Lee, I., Jeong, S., Yeo S., & Moon, J. (2012). A novel method for SQL injection attack detection based on removing SQL query attribute values. *Mathematical and Computer Modelling, 55,* 58-68.

[5] Shar, L. K., & Kuan, H. B. (2012). Mining input sanitization patterns for predicting SQL injection and cross site scripting vulnerabilities. *Proceedings of the 34th Int'l Conference on Software Engineering* (pp. 1293-1296). Zurich, Switzerland.

[6] Bisht, P., Madhusudan, P., & Venkatakrishnan, V. N. (2010). CANDID: Dynamic candidate evaluations for automatic prevention of SQL injection attacks. *ACM Trans. on Information and System Security, 13(2).*

[7] Pomeroy, A., & Tan, Q. (2011). Effective SQL injection attack reconstruction using network recording.

Proceedings of the 11th IEEE Int'l Conference on Computer and Information Technology (pp. 552-556). Paphos, Cyprus.

[8] Avireddy, S., Perumal, V., & Gowraj, N., *et al.* (2012). Random4: An application specific randomized encryption algorithm to prevent SQL injection. *Proceedings of the 2012 IEEE 11th Int'l Conference on Trust, Security and Privacy in Computing and Communications* (pp. 1327-1333). Liverpool, UK.

[9] Balasundaram, I., & Ramaraj, E. (2011). An authentication mechanism to prevent SQL injection attacks. *International Journal of Computer Applications, 19(1),* 30-33.

[10] Ali, S., Rauf, A., & Javed, H. (2009). SQLIPA: An authentication mechanism againest SQL injection. *European Journal of Scientific Research*, *38(4),* 604-611.

[11] Kindy, D. A., & Pathan, A. K. (2011). A survey on SQL injection: vulnerabilities, attacks, and prevetion techniques. *Proceedings of the 15th IEEE Symposium on Consumer Electronics*. Singapore.

[12] Srivastava, S. (2012). A survey on: Attacks due to SQL injection and their prevention method for web application*. International Journal of Computer Science and Information Technologies, 3(1),* 3225-3228.

[13] PHP. Retrieved March 10, 2014, from http://www.php.net/manual/en/function.hash.php

[14] SHA-2. Retrieved March 10, 2014, from http://en.wikipedia.org/wiki/SHA-2.

Yogesh Bansal is a graduate student in the Department of Computer Science at California State University, Fresno, CA USA. He received his B.Tech degree in electronics and communication engineering from Kurukshetra University, India in 2005 and worked in the IT Industry, MindTree and Samsumg, thereafter for six years. His research interests include cloud computing, database and network security.

Jin H. Park is an assistant professor in the Department of Computer Science at California State University, Fresno, CA, USA. He received his Ph.D. degree in computer science from Oklahoma State University, Stillwater, OK, USA in 1998. His research interests include high performance computing, parallel and distributed processing, bioinformatics and computational biology, network and reconfigurable computing.