Protect Web Applications against SQL Injection Attacks Using Binary Evaluation Approach

Venkatramulu Sunkari
Assoc.Prof.CSE Department
Kakatiya Institute of Technology and Science Warangal

Dr.C.V.Guru rao
Professor in CSE Dept., Dean Evaluation
SR Engineering College, Warangal

Abstract—today's across the globe, the data will access possibly in distant places through internet. Therefore even a unauthorized folks can access the data. The unauthorized access to data attack increases every day. SQL injection Attacks (SQLIAs) are probably the most significant associated with such web application risks. SQL shot takes advantage of the syntax of SQL to be able to inject commands that will read or maybe modify a database or compromise, this is of the main query. The product range of this kind of attacks is frequently disastrous and range from leaking associated with sensitive information to trashing customer facts. How complete people think that their information is stored safe and without any attacks. Detecting and preventing the attacks is a challenging task. The existing scenario is a highly automated approach pertaining to protecting Net applications through SQLIAs. Our approach involves certain things including identifying reliable data, making use of dynamic tainting to be able to track reliable data, and permitting only reliable data to the semantically relevant parts of queries including SQL keywords and workers. Unlike preceding approaches dependant on dynamic tainting, our technique is based on positive tainting and binary evaluation, which clearly identifies reliable (rather in comparison with untrusted) data in the program. Using this method, we get rid of the problem associated with false negatives that could result from the incomplete identification of most untrusted information sources. Our strategy against SQLIAs is based on dynamic tainting, that's previously been utilized to address security problems linked to input validation. Traditional energetic tainting strategies mark a number of untrusted information (typically end user input) while tainted, trail the stream of tainted because data with runtime, which will help prevent this information from used in potentially harmful ways.

Keywords— SQL Injection, Security, web application

I. INTRODUCTION

Internet applications usually are applications that could be accessed on the internet by using any compliant Browser that works on any OS.They have grown ubiquitous due to the convenience, versatility, availability, and interoperability them to provide. Web programs interface together with databases which contain information, for example customer labels, preferences, bank card numbers, buy orders, and so on. Web programs build SQL queries to gain access to these databases, simply, on user-provided data. The objective is in which Web programs will restriction the types of queries that could be generated to your safe subset of all possible inquiries, regardless involving what enter users present. However, inadequate entered data validation can easily enable attackers to get complete use of such databases. Attackers can easily exploited these vulnerabilities and submit strings which contain specially encoded database commands. Once the Web application builds the query by making use of these strings and submits your query to its actual database, the attacker's embedded commands usually are executed with the database and also the attack works. The outcomes of these attacks tend to be disastrous which enable it to range from leaking involving sensitive info (for instance, customer data) towards destruction involving database articles. In this paper, we propose a highly automatic approach intended for dynamic detection and avoidance of SQLIAs.Our approach works by identifying "trusted" strings in an application in addition to allowing these trustworthy strings to become used to create the semantically relevant areas of a SQL query for example keywords or operators.

.Web applications are also vulnerable to a variety of new threats. SQL injection Attacks (SQLIAs) are the most significant involving such risks. SQLIA s happens to be increasingly repeated and postures very considerable security risks simply because they can give attackers unrestricted use of the databases that underlie

Net applications. SQL hypodermic injection takes benefit from the format of SQL to inject commands that will read or modify the database, or compromise this original query. SQLIAs undoubtedly are a class involving code hypodermic injection attacks that take advantage of the lack of validation user input data. These attacks occur while developers incorporate hard-coded strings with user-provided input to create dynamic inquiries. Intuitively, if user input seriously isn't properly validated, attackers may be able to change developer's intended SQL demand by placing new SQL keywords and phrases or staff through exclusively crafted enters strings.

In This paper we introduced a highly automatic approach intended for protecting Net applications from SQLIAs. Our approach involves 1) identifying trusted info sources in addition to marking data originating from these options as trustworthy, 2) using dynamic tainting to track trustworthy data with runtime, and 3) permitting only trustworthy data to make the semantically relevant areas of queries for example SQL keywords and phrases and staff. Unlike past approaches dependant on dynamic tainting, our technique will be based upon positive tainting, which explicitly identifies trustworthy (rather than untrusted) data in a program. In this way, we eliminate the problem involving false negatives that will result from the incomplete identification of all untrusted information sources.

Our approach against SQLIAs will be based upon dynamic tainting, which includes previously been accustomed to address safety problems in connection with input affirmation. Traditional vibrant tainting methods mark selected untrusted information (typically user input) as tainted, track the move of reflectivity of the good data with runtime, preventing this information from getting used in perhaps harmful approaches. Our approach makes various conceptual in addition to practical advancements over conventional dynamic tainting methods by using the attributes of SQLIAs in addition to Web programs. First, contrary to existing vibrant tainting techniques, our approach will be based upon the novel reasoning behind positive tainting, that's, the id and marking of trustworthy, instead involving untrusted, information . second, it executes syntax-aware evaluate of query strings before these are sent towards database in addition to blocks almost all queries whoever nonliteral elements (that is actually, SQL keywords and phrases and operators) contain a number of characters with no trust marks.

II. RELATED WORK

The use of dynamic tainting to prevent SQLIAs has been investigated by several researchers. Two approaches are most similar to our approach, they are Nguyen-Tuong et all[21]. and Pietraszek[22] and Berghe[3]. Similarly, we track taint information at the character level and use a syntax-aware evaluation to examine tainted input. However, our approach differs from theirs in several important aspects. First, our approach is based on the novel concept of positive tainting, which is an inherently safer way of identifying trusted data. Second, we improve on the idea of syntax-aware evaluation by 1) using a database parser to interpret the query string before it is executed, thereby ensuring that our approach can handle attacks based on alternate encodings, and 2) providing a flexible mechanism that allows different trust policies to be associated with different input sources. Finally, a practical advantage of our approach is that it has more lightweight deployment requirements. Their approaches require the use of a customized PHP runtime interpreter, which adversely affects the portability of the approaches.

Other dynamic tainting approaches more loosely related to our approach are those by Haldar et all [9]. And Martin et all [20]. Although they also propose dynamic tainting approaches for Java-based applications, their techniques significantly differ from ours. First, they track taint information at the level of granularity of strings, which introduces imprecision in modelling string operations. Second, they use declassification rules, instead of syntax-aware evaluation, to assess whether a query string contains an attack. Declassification rules assume that sanitizing functions are always effective, which is an unsafe assumption and may leave the application vulnerable to attacks. In many cases, attack strings can pass through sanitizing functions and may still be harmful. Another dynamic tainting approach, proposed by Newsome and Song, focuses on tainting at a level that is too low to be used for detecting SQLIAs and has a very high execution overhead. Xu et all[23] propose a generalized tainting mechanism that can address a wide range of input validation- related attacks, targets C programs, and works by instrumenting the code at the source level. Their approach can be considered a framework for performing dynamic taint analysis on C programs. As such, it could be leveraged to implement a version of our approach for C-based Web applications.

Researchers also proposed dynamic techniques against SQLIAs that do not rely on tainting. These techniques include Intrusion Detection Systems (IDSs) and automated penetration testing tools. Scott and Sharp propose Security Gateway, which uses developer-provided rules to filter Web traffic, identify attacks, and apply preventive transformations to potentially malicious inputs. The success of this approach depends on the ability of developers to write accurate and meaningful filtering rules. Similarly, Valeur et all[24]. Developed IDS that uses machine learning to distinguish legitimate and malicious queries. Their approach, like most learning-based techniques, is limited by the quality of the IDS training set. Machine learning was also used in WAVES, an automated penetration testing tool that probes Web sites for vulnerability to SQLIAs. Like all testing tools, WAVES cannot provide any guarantees of completeness. SQLrand appends a random token to SQL keywords and operators in the application code. A proxy server then checks to make sure that all keywords and operators contain this token before sending the query to the database. Because the SQL keywords and operators injected by an attacker would not contain this token, they would be easily recognized as attacks. The drawbacks of this approach are that the secret token could be guessed, thus making the approach ineffective, and that the approach requires the deployment of a special proxy server [1].

Dynamic tainting has also been successfully used in the context of information-flow security to enforce information-flow policies. Such policies define limits on how information is used within a system. An example of information-flow security policy in the physical world is a military system where classified information is forbidden to be transferred to individuals without the appropriate clearance level. Dynamic tainting is an ideal technique for enforcing information-flow policies in a software system; different taint markings can be used to label sensitive information and then the analysis can check whether marked data reaches parts of the system that it is not supposed to reach according to the policies in place. Also in this case, there are several variations of this general approach. The RIFLE system provides architecture-based support for information security by tracking explicit and implicit information flows. Chow and colleagues present TAINTBOCHS, a simulator that can track tainted data through an entire system, including hardware, operating system, and applications. Using their system, they investigate the data lifetime of sensitive information in several commonly-used applications. Finally, McCamant and Ernst present a technique that produces an upper bound of the amount of information leaked by a program at runtime [2].

Model-based approaches against SQLIAs include AMNESIA, SQL-Check, and SQLGuard. AMNESIA, previously developed by two of the authors, combines static analysis and runtime monitoring to detect SQLIAs. The approach uses static analysis to build models of the different types of queries that an application can generate and dynamic analysis to intercept and check the query strings generated at runtime against the model. Queries that do not match the model are identified as SQLIAs. A problem with this approach is that it is dependent on the precision and efficiency of its underlying static analysis, which may not scale to large applications. Our new technique takes a purely dynamic approach to preventing SQLIAs, thereby eliminating scalability and precision problems. SQLCheck identifies SQLIAs by using an augmented grammar and distinguishing untrusted inputs from the rest of the strings by means of a marking mechanism. The main weakness of this approach is that it requires the manual intervention of the developer to identify and annotate untrusted sources of input, which introduces incompleteness problems and may lead to false negatives. Our use of positive tainting eliminates this problem while providing similar guarantees in terms of effectiveness. SQLGuard is an approach similar to SQLCheck. The main difference is that SQLGuard builds its models on the fly by requiring developers to call a special function and to pass to the function the query string before user input is added.

Other approaches against SQLIAs rely purely on static analysis. These approaches scan the application and leverage information flow analysis or heuristics to detect code that could be vulnerable to SQLIAs. Because of the inherently imprecise nature of the static analysis that they use, these techniques can generate false positives. Moreover, since they rely on declassification rules to transform untrusted input into safe input, they can also generate false negatives. Wassermann and Su propose a technique that combines static analysis and automated reasoning to detect whether an application can generate queries that contain tautologies. This technique is limited, by definition, in the types of SQLIAs that it can detect. Finally, researchers have investigated ways to statically eliminate vulnerabilities from the code of a Web application. Defensive coding best practices have been proposed as a possible approach, but they have limited effectiveness because they rely almost exclusively on the ability and training of developers. Moreover, there are many well-known ways to evade some defensive-coding practices, including “pseudo remedies” such as stored procedures and prepared statements. Researchers have also developed special libraries that can be used to safely create SQL queries. These approaches, although highly effective, require developers to learn new APIs, can be very expensive to apply on legacy code, and sometimes limit the expressiveness of SQL.

III. PROPOSED WORK

In our proposed system ,protect the web applications using binary evaluation for the following SQL injection attacks.

A. Tautologies

Tautology-based attacks are among the simplest and best known types of SQLIAs. The general goal of a tautology based attack is to inject SQL tokens that cause the query’s conditional statement to always evaluate to true. Although the results of this type of attack are application specific, the most common uses are bypassing authentication pages and extracting data. In this type of injection, an attacker exploits a vulnerable input field that is used in the queries WHERE conditional.

This conditional logic is evaluated as the database scans each row in the table. If the conditional represents a tautology, the database matches and returns all of the rows in the table as opposed to matching only one Row, as it would normally do in the absence of injection.

B. Union Queries

Union queries are a more sophisticated type of SQLIA that can be used by an attacker to achieve this goal, in that they cause otherwise legitimate queries to return additional data. In this type of SQLIA, attackers inject a statement of the form “UNION < injected query >.” By suitably defining < injected query >, attackers can retrieve information from a specified table. The outcome of this attack is that the database returns a data set that is the union of the results of the original query with the results of the injected query.

C. Piggybacked Queries

Similar to union queries, this kind of attack appends additional queries to the original query string. If the attack is successful, the database receives and executes a query string that contains multiple distinct queries. The first query is generally the original legitimate query, whereas subsequent queries are the injected malicious queries. This type of attack can be especially harmful because attackers can use it to inject virtually any type of SQL command..

D. Malformed Queries

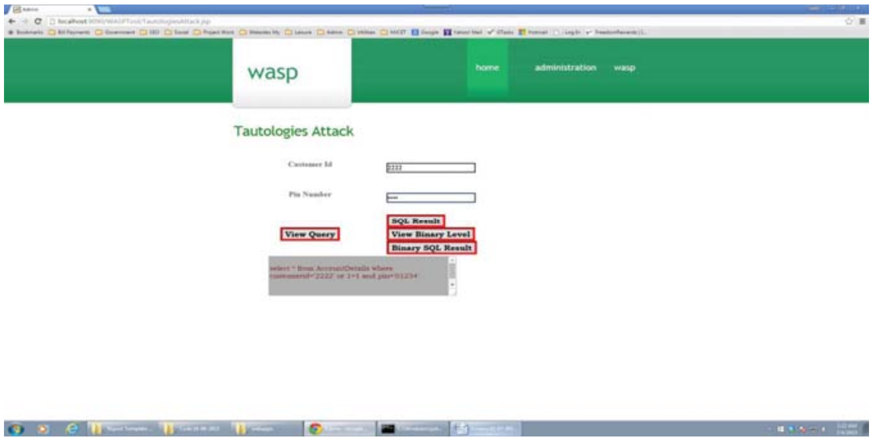
Union queries and piggybacked queries let attackers perform specific queries or execute specific commands on a database, but require some prior knowledge of the database schema, which is often unknown. Malformed queries allow for overcoming this problem by taking advantage of overly descriptive error messages that are generated by the database when a malformed query is rejected. When these messages are directly returned to the user of the Web application, instead of being logged for debugging by developers, attackers can make use of the debugging information to identify vulnerable parameters and infer the schema of the underlying database.

E. Alternate Encodings

Many types of SQLIAs involve the use of special characters such as single quotes, dashes, or semicolons as part of the inputs to a Web application. Therefore, basic protection techniques against these attacks check the input for the presence of such characters and escape them or simply block inputs that contain them. Alternate encodings let attackers modify their injected strings in a way that avoids these typical signature-based and filter-based checks. Encodings such as ASCII, hexadecimal, and Unicode can be used in conjunction with other techniques to allow an attack to escape straightforward detection approaches that simply scan for certain known “bad characters.

IV. RESULTS

The concept of this paper have been implemented and we got good and efficient results which have been efficiently tested on different Datasets are shown below fig 1,2,3&4. The proposed paper work is implemented in Java technology on a Pentium-IV PC with minimum 20 GB hard-disk and 1GB RAM.

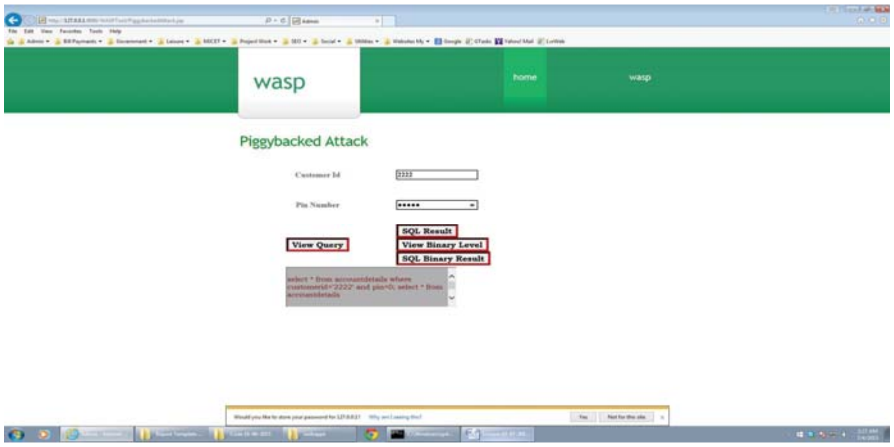


Customer Id 2222

Pin Number

```
select * from AccountDetails where
customerid='2222' or 1=1 and pin=01234
```

Fig. 1 Computing Tautological Attack

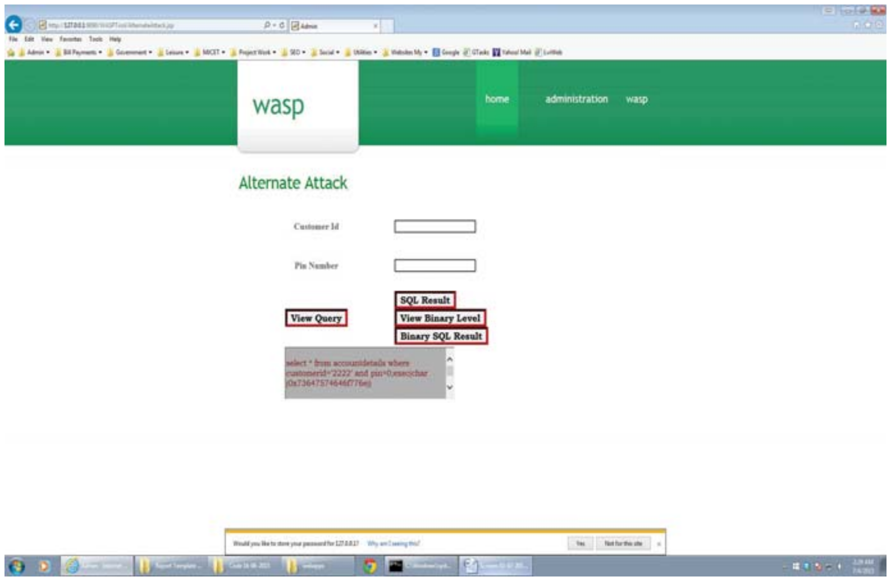


Customer Id 2222

Pin Number •••••

```
select * from accountdetails where
customerid='2222' and pin=0; select * from
accountdetails
```

Fig. 1 Computing Piggybacked Attack

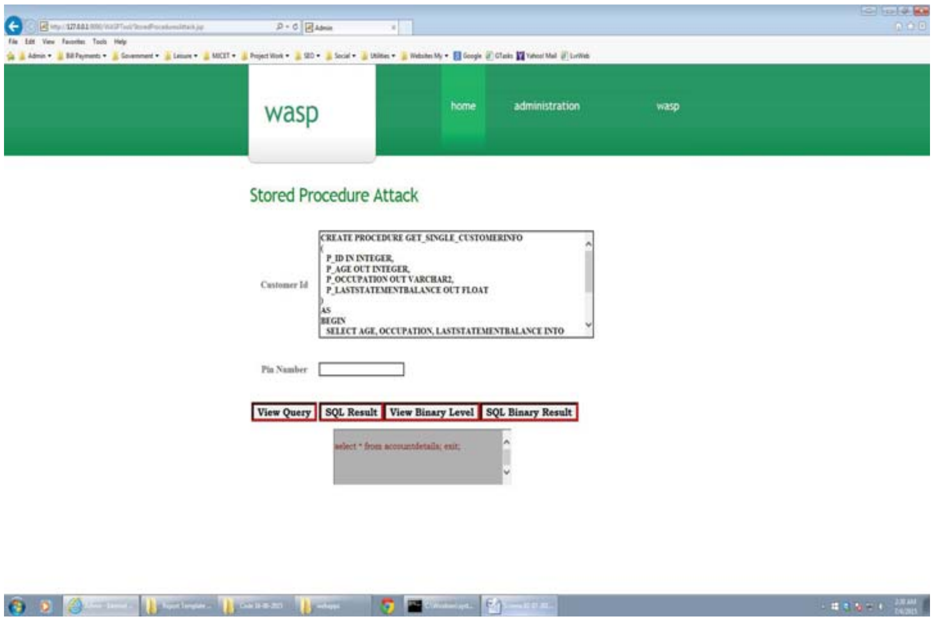


Customer Id

Pin Number

```
select * from accountdetails where
customerid='2222' and pin=0;exec(char
(0x73647574646f776e))
```

Fig. 3 Computing Alternate Attack



Stored Procedure Attack

Customer Id

```
CREATE PROCEDURE GET_SINGLE_CUSTOMERINFO
(
  P_ID IN INTEGER,
  P_AGE OUT INTEGER,
  P_OCCUPATION OUT VARCHAR2,
  P_LASTSTATEMENTBALANCE OUT FLOAT
)
AS
BEGIN
  SELECT AGE, OCCUPATION, LASTSTATEMENTBALANCE INTO
```

Pin Number

```
select * from accountdetails; exit;
```

Fig. 4 Computing Stored Procedure Attack

V. Conclusions

We have presented a new novel, extremely automated method for sensing and blocking SQL injection attacks in Web applications. Our basic approach is made of Identifying dependable data options and tagging data coming from these options as dependable, Using dynamic tainting we are able to track dependable data at runtime, and Making it possible for only dependable data being SQL keywords and phrases or staff in problem strings. Unlike previous approaches based on dynamic tainting, our technique is based on positive tainting, which explicitly identifies dependable (rather in comparison with untrusted) data within the program. In this way, we eliminate the problem of false negatives that may result from incomplete identification of untrusted information sources. False positives effortlessly eliminated while in testing using our approach. Our method also given more practical advantages above the many current techniques whoever application requires customized and complex runtime conditions. The method is defined with the application levels, requires not any modification in the runtime system, and imposes a low execution cost. We get evaluated our approach by creating a prototype software, and using this software safeguard several applications when suffering from a significant and varied pair of attacks and legitimate accesses. Using our approach protect web applications in a better way.

References

[1] C. Anley, "Advanced SQL Injection In SQL Server Applications," white paper, Next Generation Security Software, 2002.

[2] S.W. Boyd and A.D. Keromytis, "SQLrand: Preventing SQL Injection Attacks," Proc. Second Int'l Conf. Applied Cryptography and Network Security, pp. 292-302, June 2004.

[3] G.T. Buehrer, B.W. Weide, and P.A.G. Sivilotti, "Using Parse Tree Validation to Prevent SQL Injection Attacks," Proc. Fifth Int'l Workshop Software Eng. and Middleware, pp. 106-113, Sept. 2005.

[4] J. Clause, W. Li, and A. Orso, "Dytan: A Generic Dynamic Taint Analysis Framework," Proc. Int'l Symp. Software Testing and Analysis, pp. 196-206, July 2007.

[5] W.R. Cook and S. Rai, "Safe Query Objects: Statically Typed Objects as Remotely Executable Queries," Proc. 27th Int'l Conf. Software Eng., pp. 97-106, May 2005.

[6] Top Ten Most Critical Web Application Vulnerabilities," OWASP Foundation, http://www.owasp.org/documentation/ topten.html, 2005.

[7] C. Gould, Z. Su, and P. Devanbu, "JDBC Checker: A Static Analysis Tool for SQL/JDBC Applications," Proc. 26th Int'l Conf. Software Eng., formal demos, pp. 697-698, May 2004.

[8] C. Gould, Z. Su, and P. Devanbu, "Static Checking of Dynamically Generated Queries in Database Applications," Proc. 26th Int'l Conf. Software Eng., pp. 645-654, May 2004.

[9] V. Haldar, D. Chandra, and M. Franz, "Dynamic Taint Propagation for Java," Proc. 21st Ann. Computer Security Applications Conf., pp. 303-311, Dec. 2005.

[10] W. Halfond, A. Orso, and P. Manolios, "Using Positive Tainting and Syntax-Aware Evaluation to Counter SQL Injection Attacks," Proc. ACM SIGSOFT Symp. Foundations of Software Eng., pp. 175-185, Nov. 2006.

[11] W.G. Halfond and A. Orso, "AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks," Proc. 20th IEEE and ACM Int'l Conf. Automated Software Eng., pp. 174-183, Nov. 2005.

[12] W.G. Halfond, J. Viegas, and A. Orso, "A Classification of SQLInjection Attacks and Countermeasures," Proc. IEEE Int'l Symp. Secure Software Eng., Mar. 2006.

[13] M. Howard and D. LeBlanc, Writing Secure Code, second ed. Microsoft Press, 2003.

[14] Y. Huang, S. Huang, T. Lin, and C. Tsai, "Web Application Security Assessment by Fault Injection and Behavior Monitoring," Proc. 12th Int'l Conf. World Wide Web, pp. 148-159, May 2003.

[15] Y. Huang, F. Yu, C. Hang, C.H. Tsai, D.T. Lee, and S.Y. Kuo, "Securing Web Application Code by Static Analysis and Runtime Protection," Proc. 13th Int'l Conf. World Wide Web, pp. 40-52, May 2004.

[16] M. Martin, B. Livshits, and M.S. Lam, "Finding Application Errors and Security Flaws Using PQL: A Program Query Language,"Proc. 20th Ann. ACM SIGPLAN Conf. Object Oriented Programming Systems Languages and Applications, pp. 365-383, Oct. 2005.

[17] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D.Evans, "Automatically Hardening Web Applications Using Precise Tainting Information,"Proc. 20th IFIP Int'l Information Security Conf., May 2005.

[18] T. Pietraszek and C.V. Berghe, "Defending against Injection Attacks through Context-Sensitive String Evaluation," Proc. Eighth Int'l Symp. Recent Advances in Intrusion Detection, Sept. 2005.

[19] W. Xu, S. Bhatkar, and R. Sekar, "Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks," Proc. 15th Usenix Security Symp., Aug. 2006.

[20] F. Valeur, D. Mutz, and G. Vigna, "A Learning-Based Approach to the Detection of SQL Attacks," Proc. Conf. Detection of Intrusions and Malware and Vulnerability Assessment, July 2005.