Runtime Monitors for Tautology based SQL Injection Attacks

Ramya Dharam and Sajjan G. Shiva
Computer Science Department
University of Memphis
Memphis, TN, USA
{rdharam, sshiva} @memphis.edu

Abstract – **Increased usage of web applications in recent years has emphasized the need to achieve (i) confidentiality, (ii) integrity, and (iii) availability of web applications. Backend database being the main target for external attacks such as SQL Injection Attacks, there is an emerging need to handle such attacks to secure stored information. Pre-deployment testing alone does not ensure complete security and hence post-deployment monitoring of web applications during its interaction with the external world can help us to handle SQL Injection Attacks in a better way. In this paper, we present a framework which can be used to handle tautology based SQL Injection Attacks using post-deployment monitoring technique. Our framework uses two pre-deployment testing techniques i.e. basis path and data flow testing techniques to identify legal execution paths of the software. Runtime monitors are then developed and integrated to observe the behavior of the software for identified execution paths such that their violation will help to detect and prevent tautology based SQL Injection Attacks.**

Keywords- Runtime Monitors, Path Testing, Data Flow Testing, Post-deployment Monitoring, Tautology, SQL Injection Attacks (SQLIAs).

I. INTRODUCTION

Web applications are used by organizations to provide services like online banking, online shopping and social networking; over the recent years our dependence on web applications has increased drastically in our everyday routine activities. So we expect these web applications to be secure and reliable when we are paying bills, shopping online, making transactions etc. These web applications consists of underlying databases containing confidential user's information like financial information records, medical information records, personal information records which are highly sensitive and valuable, which in turn makes web applications an ideal target for attacks. One such type of attack, SQL Injection Attacks (SQLIAs), is one of the major security threats to web applications [1]. This attack will give attackers access to the database underlying the web applications and also the rights to retrieve, modify and delete confidential user information stored in the database resulting in security violations, identity theft, etc.

SQLIAs occur when data provided by the user is included directly in a SQL query and is not properly validated. Attackers take advantage of this improper input validation and submit input strings that contain specially encoded database commands. When the application builds a query using these strings and submits the query to its underlying database, the attacker's embedded commands are executed by the database and the attack succeeds [2].

It has been found that inadequate input validation performed within an application is the major cause for SQLIAs. But, relying on input validation techniques alone for defending the application against SQLIAs is problematic and also insufficient to achieve complete security of the application. Although implementing input validation routines can serve as a first level of defense, they cannot defend against sophisticated attack techniques that inject malicious inputs into SQL queries [2, 3]. Tools such as firewalls and Intrusion Detection Systems (IDSs) are ineffective against SQLIAs, because ports which are open in firewalls for regular web traffic in the application level are used to perform SQLIAs. A variety of programming practice guidelines and web application security testing tools and scanners have also been proposed by the research community to detect and prevent SQLIAs. Inspite of implementing the mentioned preventive techniques attackers are still able to successfully perform SQLIAs on web applications and get access to the confidential user information.

In this paper, we introduce a framework to develop runtime monitors for performing post-deployment monitoring of the application to detect and prevent tautology based SQLIAs. The proposed framework is an extension of our framework proposed in [4] to detect and prevent path traversal attack based on behavior of the software application. The framework proposed in this paper uses two pre-deployment testing techniques to help in the development of runtime monitors.

The paper is organized as follows. In Section 2, we present our framework. In Section 3, we discuss the implementation of our proposed framework to detect tautology based SQLIAs in a Java application and the results obtained. In Section 4, we present Game Inspired Defense Architecture (GIDA) framework. Section 5 discusses the related work and conclusion is discussed in Section 6.

II. PROPOSED FRAMEWORK

Our proposed framework is an extension of our previous work in [4] that introduced a post-deployment monitoring technique to handle path traversal attack. In this section, we propose a framework to handle tautology based SQLIAs in Java applications using post-deployment monitoring technique.

The basic idea behind our proposed framework is that (1) the source code contains certain critical variables that interact with the external world by accepting user inputs, build queries and process them by accessing the internal database (2) monitor the behavior of application during its execution with respect to the indentified critical variable to detect and prevent tautology based SQLIAs.

Our proposed framework first uses a software repository which consists of a collection of documents related to requirements, security specifications, source code, etc. to find the critical variables. Then, a combination of basis path and data flow testing techniques is used to find all the legal/valid execution paths the critical variables can take during their lifetime in the application. Runtime monitors are then developed to observe the path taken by the critical variables and check them for compliance with the obtained legal paths. During runtime, if the path taken by the identified critical variable violates the legal paths obtained, implies that the critical variable consists of the malicious input from the external user and the query formed is trying to access confidential information from the backend database. This abnormal behavior of the application due to the critical variables is identified by the runtime monitor and immediately notifies to the administrator. The framework described is shown in Figure 1 and consists of three main steps which are discussed below in detail.

Identification of critical variables: Scan the software repository to identify all the critical variables present in the source code. Critical variables are those which interact with the external world by accepting user input, and also which are part of critical operations that involve query executions. For each of the critical variables identified, checkpoint i.e. snippets of code which verify the values returned by the query executions are inserted. By doing so, malicious inputs from the external users that lead to SQLIAs can be easily detected based on the results returned by the query execution.

Build legal execution paths: By combining data flow and basis path testing, legal execution paths of the application are obtained. Data flow testing of the critical variables help in identification of all the legal sub paths that can be taken by critical variables during the execution. Basis path testing is performed to identify the minimum number of legal execution paths of the software. Since basis path testing leads to reduced number of monitorable paths, the complexity of our proposed technique in terms of integrating monitors across multiple paths also reduces. The path identification function

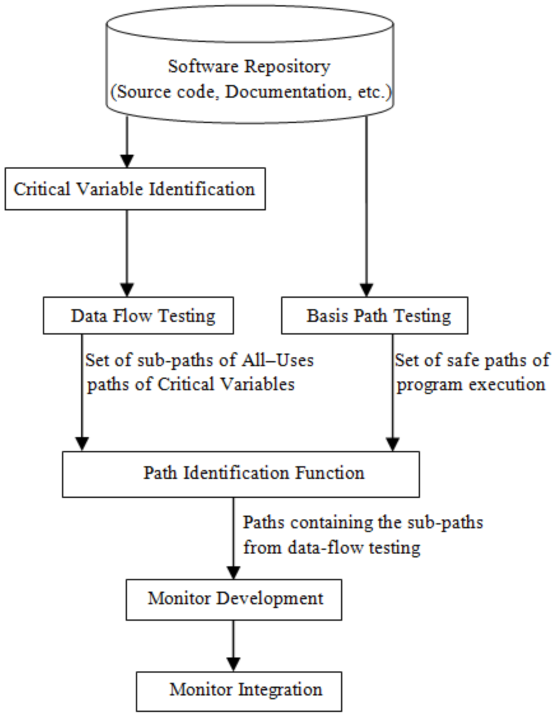

Figure 1. Runtime monitoring framework for tautology based SQLIAs.

builds the set of critical paths to be monitored in the application to detect and prevent tautology based SQLIAs.

Let $C = \{C^1, C^2 \ldots\ldots, C^m\}$ be a set of *m* critical variables identified during critical variable identification phase. Let $P_C = \{\{ P_C^1 \} \cup \{ P_C^2 \} \cup \ldots\ldots, \{ P_C^m \}\}$ be a set of critical variable sub paths such that, P_C^i is a set of all valid sub paths a critical variable C^i can take during its lifetime in the software, $i \in [0, m]$ and is identified by performing data flow testing on C^i. Let $P = \{P^1, P^2 \ldots\ldots, P^k\}$ be a set of k legal paths identified using basis path testing and CP is a set of paths we intend to monitor. CP is identified using the pseudo code shown below:

```
CP = { }
for every P^j ϵ P and
      for every P_C^i ϵ P_C
            if (P^j ∩ P_C^i == P_C^i)
                CP = CP U { P^j }
where, i ϵ [0, m] and j ϵ [0, k].
```

We thus identify all the critical paths of the software to be monitored.

Runtime monitoring: In this phase, we map the identified critical paths to regular expressions and use the monitoring oriented programming (MOP) [5] tool to generate monitors. The generated monitor is then integrated with the respective module of the application for monitoring the critical paths. Henceforth, on every query execution, the runtime monitor tracks the identified critical variable by monitoring their execution path, and also verifies the results returned with respect to the instrumented checkpoint. When a critical variable violates the checkpoint and in turn follows an invalid path, the runtime monitor immediately detects the abnormal behavior of the application due to the critical variable and notifies the administrator.

III. AN EXAMPLE

In this section, we introduce an example of a Java application that is vulnerable to tautology based SQLIAs and explain how our proposed framework can be used to detect and prevent the attack. This particular example illustrates an attack based on injecting a tautology into the query string.

The Java application developed uses a XAMPP web server for Apache HTTP service and MySQL database for the back end storage of data. The application simulates the working of an employee information retrieval website, which uses a back end database to store the employee related information and each record is unique to a single employee. Figure 2 shows a Java code snippet of the employee information retrieval application as described above. First, a method inputUserInfo() is invoked to accept the login information from the user which includes both username and password. The submitted credentials are then used to dynamically build the query1 as shown below:

String query1 = "Select * FROM personalinfo where username = '"+strLine1+"' and password = '"+strLine2+"'";

During the execution of application, SQL query string as formed above will be submitted to the database. The response received from the database consists of all records satisfying the SQL query. Any website that uses this code would be vulnerable to SQL Injection Attacks when a user enters "' OR 1=1 --" and "", instead of "John" and "Pouch2345", the resulting query is:

SELECT * FROM userInfo WHERE login='' OR 1=1 --' AND pass=' ';

The character "--" indicates the beginning of a comment, and everything following the comment is ignored. The database interprets everything after the WHERE token as a conditional statement, and inclusion of the "OR 1=1" clause turns this conditional into a tautology. Thus, when the above query is executed, more than one record is returned by the database. As a result, the information about all the users will be displayed by the application. In this way, an attacker could insert a wide range of SQL commands via this exploit.

We now discuss the implementation of our proposed framework which is an extension of our previous work in [4] to handle tautology based SQLIAs existing in the Java application described above.

Using the software repository as explained earlier, query1 is identified as one of the critical variables, because it embeds the inputs received from the external user and holds the results of the query execution by interacting with the internal database. The checkpoint to this variable is instrumented in the source code, which checks the number of records returned by the query execution which will help us to detect the malicious inputs.

```
public void inputUserInfo()
{
        System.out.println("Enter the username");
        Scanner in = new Scanner(System.in);
        strLine1 = in.nextLine();
        System.out.println("Enter the password");
        Scanner in = new Scanner(System.in);
        strLine2 = in.nextLine();
}
ob1.inputUserInfo();
//generate query to send to database
String query1 = "Select * FROM userinfo where username = '"+strLine1+"'
and password = '"+strLine2+"'";
Class.forName("com.mysql.jdbc.Driver");
//connect to the database and send the query
Connection con = DriverManager.getConnection(dbUrl,username,password);
con.setAutoCommit(false);
Statement stmt = con.createStatement();
ResultSet resultset = stmt.executeQuery(query1);
while(resultset.next())
{rowCount++;}
if(rowCount == 1)
{
        ResultSet resultset1 = stmt.executeQuery(query1);
        con.commit();
        while(resultset1.next())
        {
                String username1 = resultset1.getString("username");
                String password1 = resultset1.getString("password");
                ob1.display(username1, password1);
        }/*end while*/
}/*end if*/
else{
        con.rollback();
        ob1.attacker();
}/*end else*/
```

Figure 2. Java code snippet.

Pre-deployment testing techniques such as data flow and basis path testing are used to obtain all valid execution paths of the application. In our framework, we consider the possible sequence of function calls that can be called upon as a valid path which reflects the valid execution of the application. The path identification function is then used to obtain all the critical paths of the application, which needs to be monitored.

The developed application will first check for the number of rows returned by the database after the execution of the query. If the number of rows returned by the application is more than one, the application will rollback the transaction occurred at that point and function named attacker() is invoked immediately. To prevent the attacker from gaining access to information about all users, the runtime monitor developed by using the proposed framework observes the behavior of the application. When an invocation to the attacker() function is made after the invocation of the inputUserInfo() function, this abnormal behavior of the application which indicates that more than one row has been returned by the database is immediately identified by the runtime monitor.

The monitor then halts the execution of the respective module in the application trying to access the internal database and notifies the administrator about the possible occurrence of tautology based SQLIAs. Figure 3 shows the monitor code instrumented to the respective module in the source code of the application.

```
SQLIAdemo{

  event inputUserInfo before(Object o):
     call(* *.inputUserInfo() && target(o) {}

  event display before(String username1, String password1):
     call(* *.display(String, String)) && args(username1,password1) {}

  event attacker after(Object o2):
     call(* *.attacker() && target(o2) {}

  ere : (inputUserInfo display)

  @fail {
              System.out.println(" ");
              System.out.println("MOP:FAIL");
              System.out.println(" ");
       }

  @match {
              System.out.println(" ");
              System.out.println("MOP:MATCH");
         }
 }
```

Figure 3. Monitor development code.

Thus, the run time monitor developed will successfully detect tautology based SQL Injection Attacks and will prevent the attacker from retrieving confidential information about all the users.

IV. GIDA FRAMEWORK

The framework proposed in the current paper will be used to enhance the working of Target Self-monitoring Application component present in the GIDA architecture as shown in Figure 4. Shiva et al. in [6, 7] proposed Game Inspired Defense Architecture (GIDA) as a holistic approach designed to secure target applications/network against probable attacks.

GIDA architecture shown in Figure 4 consists of the following components, namely: Intrusion Detection System (IDS), Knowledge Management System (KMS), Game Inspired Decision Module (GIDM), Honeypot and Target Self Monitoring Application (TSMA). In rest of the paper, we refer TSMA as Monitoring System (MS).

The IDS is used to monitor the network traffic and capture information about the network behavior and its usage statistics. An IDS is considered as a network level sensor in the GIDA Framework.

We consider software applications developed in modular fashion consisting of different modules/components represented using C_1, C_2, and C3 as shown in Figure 4. Different modules/components execute either in different target systems or on a single target system connected to internet. Each module/component is instrumented with their respective runtime monitor, represented using m_1, m_2 and m_3 as shown in Figure 4. The runtime monitor will detect the abnormal behavior of the module/component during its execution which may be caused due to existing internal errors

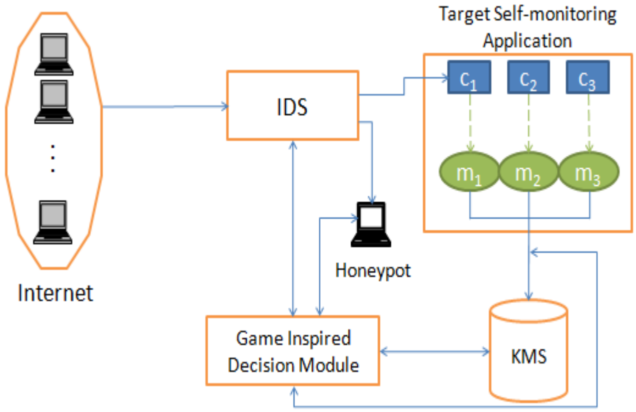

Figure 4. Game Inspired Defense Architecture (GIDA) [6]

or external attacks. The process of identifying the abnormal behavior by the runtime monitor will be performed by utilizing the framework proposed in the current paper. The runtime monitor is considered to be an application level sensor for the GIDA Framework.

Once the abnormal behavior is identified, the runtime monitor can perform any of the following functions:
i) If the monitor is able to identify the attack that has occurred, it will immediately execute defense action by itself to secure the module/component. The simplest and direct defense action to implement by the runtime monitor is to immediately halt the execution of the entire module/component depicting the abnormal behavior.
ii) If the monitor is not able to identify the attack that has occurred, then it forwards the information gathered related to the abnormal behavior to both KMS and GIDM, to identify the attack occurred and wait for the response.

The output from either the network or the application level sensors is forwarded as input to GIDM which is the brain and an important component of the proposed Game Inspired Defense Architecture (GIDA). GIDM processes the input information received from sensors (IDS and MS) and could take either of the following mentioned actions based on the input received and its knowledge about the attacks occurred:
i) If GIDM is able to identify the attack occurred, then it suggests a defense.
ii) If GIDM is not able to identify the attack occurred, it forwards the input received either from IDS or MS about the occurred abnormal behavior to KMS, for identification of the attack that has occurred and a suitable defense action to be executed.

iii) If neither KMS nor GIDM is able to identify the attack occurred, then GIDM invokes the honeypot which is primarily used for analyzing traffic and gathering additional information from the attacker.

The KMS focuses on determining the type of attack and it consists of game models mapped to the kinds of attack they can address. The KMS is based on a cyber-attack taxonomy called AVOIDT [8]. Based on the parameters provided as inputs from GIDM, KMS uses AVOIDIT to identify the characteristics of an attack. Once an attack is identified, a candidate game model which can defend against such an attack is selected and notified to GIDM. GIDM then either executes the suggested game model as the defense action to protect the target network/application or GIDM will forward the defense action information to either IDS or MS for them to execute the suggested defense action.

GIDA follows the below steps to provide security to either the target application/network:
1. Receive inputs from either the network or application sensors.
2. Identify the attack occurred either by its prior knowledge or with the help of KMS or honeypot.
3. Select a relevant game model for the occurred attack by either utilizing its prior knowledge or with the help of KMS.
4. Execute the selected game model or allow the sensors to perform defense actions to secure target application/network against the attacker.

V. RELATED WORK

In this section, we provide a survey of various existing techniques which include both static and dynamic techniques to handle tautology based SQLIAs.

In [10], Wasserman and Su propose a static analysis framework that operates directly on the source code of the application. Static analysis is used to obtain a set of SQL queries that a program may generate as a finite state automaton. The framework then applies an algorithm on the generated automaton to check whether there is a tautology and the existence of a tautology indicates the presence of a potential vulnerability. Our proposed approach detects tautology based SQL Injection Attacks based on the behavior of the application during its execution and no finite automaton is used.

In [11], Huang et al propose a web application security assessment framework called WAVES (Web Application Vulnerability Scanner). WAVES is a black box testing tool from the research community which can be used to identify web application vulnerabilities. AppScan, WebInspect and ScanDo are some of the commercially available web application back box testing tools. In practice, testing tools are useful for finding vulnerabilities but, they cannot be used to make security guarantees. While our proposed approach uses the information gathered from the testing techniques to help in the development of runtime monitors, to detect tautology based SQL Injection Attacks from observing the behavior of the application during its execution.

In [12], Valeur et al propose an Intrusion Detection System to detect SQL Injection Attacks. The proposed system uses an anomaly detection approach to learn profiles of the normal database access using different models performed by web applications. During training phase, profiles are learned automatically by analyzing a number of sample database accesses. During detection phase, anomalous queries that lead to SQL Injection Attack are identified. In our proposed approach we do not maintain profiles of database access and based on the behavior of the software during its execution detect if the application is vulnerable to SQL Injection Attack and immediately stop the execution of the software and notify the administrator about the possible exploitation of the vulnerability.

In [13], Su et al propose SQL-Check which is a runtime checking system. The approach used in SQL check will first track the user input substring in the program and syntactically track those substrings using a syntactic policy. This will specify all the permitted syntactic forms. This process forms an annotated query also called an augmented query. A parser is then used by SQL Check to parse the augmented query and to find whether the query is legitimate or not. If the query parses successfully, then the query is supposed to have met the syntactic constraints and is considered as legitimate. But, if the query has not successfully passed by the parser then it is considered to be a command injection attack query. In our proposed approach we also try to detect SQL Injection Attack at the runtime but, neither a parser nor policy is used and the SQL Injection Attacks are identified based on the anomalous behavior of the software during its execution.

In [14], Halfond et al propose a tool called Analysis and Monitoring for Neutralizing SQL Injection attacks (AMNESIA). It consists of a static and a dynamic phase. During the static phase models for the different types of queries which an application can legally generate at each point of access to the database are built. During the dynamic phase queries are intercepted before they are sent to the database and are checked against the statically built models. If the queries violate the model then a SQL Injection Attack is detected and further queries are prevented from accessing the database. Our proposed approach does not consist of a static and dynamic phase. SQL Injection attacks are detected based on the behavior of the application with the help of runtime monitors developed by using our proposed framework.

In [15], Bisht et al propose Candidate evaluations for Discovering Intent Dynamically (CANDID) which at each SQL query location dynamically mines programmer intended query structures and detects attacks by comparing it against the structure of the actual query issued. Program

Transformation is used by CANDID to retrofit web applications written in Java. In [16], Cova et al propose an approach for the anomaly based detection of state violations in web applications and designed a tool called Swaddler. The internal state of a web application is analyzed and the relationships between the applications critical execution points and the applications internal state are learned. This process of analysis and learning is used by Swaddler to identify attacks that attempt to bring an application in an inconsistent anomalous state. In our proposed approach we identify the critical variables and determine the paths to be monitored to identify the anomalous behavior of the application during its execution which will help us to detect and prevent tautology based SQLIAs.

VI. CONCLUSION

In this paper, we proposed a framework for development of runtime monitors used to perform post-deployment monitoring of the software to detect and prevent tautology based SQLIAs. Thus using our proposed framework we ensure that the quality and security of software is achieved not only during its pre-deployment phase but, also during its post-deployment phase and any possible exploitation of vulnerability in the software by an external attacker is detected and prevented. We further intend to automate the entire process of using the proposed framework to develop the runtime monitors and also extend the framework to detect and prevent the other types of attacks.

VII. REFERENCES

[1] OWASP – Open Web Application Secuirty Project. Top ten most web application vulnerabilities. http: //ww.owasp.org/index.php/OWASP_TOP_Ten_Project, April 2010.

[2] W. G. J. Halfond, A. Orso and P. Manolios, "Using Positive Tainting and Syntax-aware Evaluation to Counter SQL Injection Attacks", In SIGSOFT'06/FSE-14: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2006.

[3] W. G. J. Halfond, J. Viegas and A.Orso, "A Classification of SQL-Injection Attacks and Countermeasures", Proceedings of the IEEE International Symposium on Secure Software Engineering, 2006.

[4] Ramya Dharam, Sajjan. G. Shiva, "A Framework for Development of Runtime Monitors", International Conference on Computer and Information Sciences (ICCIS), Kuala Lumpur, Malaysia, June 2012.

[5] F. Chen and G. Rosu, "Java MOP: A Monitoring Oriented Programming Environment for Java", in Proceedings of Eleventh International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), 2005.

[6] S. Shiva, H. Bedi, C. Simmons, M. Fisher, R. Dharam, "A Holistic Game Inspired Defense Architecture", International Conference on Data Engineering and Internet Technology (DEIT), March 2011.

[7] S. Shiva, S. Roy and D. Dasgupta, "Game Theory for Cyber Security", 6th Cyber Security and Information Intelligence Research Workshop, April 2010.

[8] C. Simmons, S. Shiva, D. Dasgupta, and Q. Wu, "AVOIDT: A Cyber Attack Taxonomy", Technical Report: CS-09-003, University of Memphis, August 2009.

[9] A. Tajpour, M. Massrum and M. Z. Heydari, "Comparison of SQL Injection Detection and Prevention Techniques", 2nd International Conference on Education Technology and Computer (ICETC), 2012.

[10] G. Wassermann and Z. Su, "An Analysis Framework for Security in Web Applications", Proceedings of the FSE Workshop on Specification and Verification of Component Based Systems (SAVCBS 2004), 2004.

[11] Y.-W. 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", Proceedings of the 12th International World Wide Web Conference (WWW 2004), 2004.

[12] F. Valeur, D. Mutz, and G. Vigna, "A Learning Based Approach to the Detection of SQL Attacks", Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA), 2005.

[13] Z. Su and G. Wassermann, "The Essence of Command Injection Attacks in web Applications", The 33rd Annual Symposium on Principles of Programming Languages (POPL 2006), 2006.

[14] W. G. Halfond and A. Orso, "AMNESIA: Analysis and Monitoring for Neutralizing SQL-Injection Attacks", Proceedings of the IEEE and ACM International Conference on Automated Software Engineering (ASE 2005), Nov 2005.

[15] P. Bisht and P. Madhusudan, "CANDID: Dynamic Candidate Evaluations for Automatic Prevention of SQL Injection Attacks", Proceedings of the 14th ACM Conference on Computer and Communications Security, 2007.

[16] M. Cova, D. Balzarotti, "Swaddler: An Approach for the Anomaly-based Detection of State Violations in Web Applications", Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID), 2007.

[17] Steve Ragan. Sony was asking for it – millions of records compromised. http://www.thetechherald.com/articles/LulzSec - Sony-was-asking-for-it- millions-of-records-compromised-(Update2), June 2011.

[18] A. Orso, "Monitoring, Analysis, and Testing of Deployed Software", Proceedings of the FSE/SDP workshop on Future of software engineering research (FoSER'10), 2010.

[19] F. S. Rietta, "Application Layer Intrusion Detection for SQL Injection", Proceedings of the 44th annual southeast regional conference (ACM-SE 44), 2006.

[20] Y. Xie and A. Aiken, "Static Detection of Secuirty Vulnerabilities in Scripting Languages", Proceedings of the 15th Conference on USENIX Security Symposium (USENIX-SS'06), 2006.

[21] V. B. Livshits and M. S. Lam, "Finding Security Vulnerabilities in Java Applications with Static Analysis", Proceedings of the 14th Conference on USENIX Security Symposium (SSYM'05), 2005.