A Framework for the Detection and Prevention of SQL Injection Attacks

Emad Shafie, Antonio Cau
Software Technology Research Laboratory (STRL)
De Montfort University, Leicester, United Kingdom
eshafie@dmu.ac.uk
acau@dmu.ac.uk

Abstract: The use of Internet services and web applications has grown rapidly because of user demand. At the same time, the web application vulnerabilities have increased as a result of mistakes in the development where some developers gave the security aspect a lower priority than aspects like application usability. An SQL (structure query language) injection is a common vulnerability in web applications; it has been classified as the most dangerous type of vulnerability according to OWASP (Open Web Application Security Project) statistics (OWASP,2010). An SQL injection vulnerability can allows the hacker or illegal user to have access to the web application's database and therefore damage the data, or change the information held in the database. This paper will discuss a framework for the detection and prevention of common types of SQL injection attacks. The framework consists of three main components; the first component will check the user input for existing attacks, the second component check for new types of attacks, and the last component will block unexpected responses from the database engine. Additionally, our framework will keep track of an ongoing attack by recording and investigating user behaviour. The framework is based on the Anatempura tool, a runtime verification tool for Interval Temporal Logic properties. Existing attacks and good/bad user's behaviours can be specified using Interval Temporal Logic. Moreover, this paper will discuss a case study where various types of user behaviour are specified in Interval Temporal Logic and show how they can be detected.

Keywords: SQL injection, user input checker, runtime verification, database observer.

1. Introduction:

The permanent availability of web applications will increase the opportunity for everyone who is looking to exploit and damage these applications for illegal purposes. The common threat against the security of web application is the widespread occurrence of different types of web application vulnerability. SQL injection is a common vulnerability used to hack web application databases by executing a malicious SQL code injected by the attacker (Fu & Qian, 2008).
An example of SQL injection attack is the following: a web application needs user input to perform its task. Accordingly, web applications usually provide a login page containing two text fields to allow the user to enter his/her user name and password. This user information will be sent to the web application database to check the user information. By submitting the user data, this data will be sent to the web application database using an SQL statement as follows:

*Select * from UserTable where username= "user_entry_name" and userpassword =”user_entry_password”*

When this SQL statement is executed, the system will return the result of the query. If the user data is ok then the web application permits the user to access other pages at the website or the user data will be rejected and the login page reloads again. However, there is another scenario where the user enters the following at the user name field user name or '1'='1' - - then the SQL statement will be as follows:

*SELECT * FROM UserTable WHERE username = "user name or '1'='1' - -"*

At this stage the database engine considers any code after the keyword WHERE as a conditional statement, and because of "or 1=1 -- " the check condition is always equal to true. So, any code or condition after the double dash will be ignored. Consequently, the attacker will have unauthorized access to this web application.

The problem of this type of attack is that it cannot be handled or controlled by a firewall or other traditional communication security approaches as the attackers can gain access to the web application through the http protocol (Fu et al, 2007). Poor validation of the user input is the main reason of SQL injection attacks. Many approaches have tried to solve and block this type of vulnerability by using several different techniques ranging from static analysis approaches(Gould

2004,X 2007) to using dynamic approaches (Kosuga et al,2007) or combining both of them in one technique (Halfond,2005). This paper will discuss a novel technique to detect and prevent SQL injection attacks at the runtime using the Anatempura tool. Our framework does this by keeping track of an ongoing attack by recording and investigating user behaviour. The paper is divided in several sections as follows, Section 2 provides related work, Section 3 provides our approach in detail by describing all the framework components, Section 4 will discuss Anatempura tool, Section 5 will illustrates our framework with help of a case study, in Section 6 we give a conclusion and discuss future work.

2. Related work:

There exist many approaches that deal with SQL injection attacks, some of these approaches are as follows: Black Box testing approach, by gathering the information about all weak points in the website by using a web crawler to detect the vulnerable points that can be indictable (Huang et al , 2003). SQL Randomisation approach, by adding integer numbers randomly to each SQL keyword that used in the query statement at that application, then during execution the application will rewrite the SQL statement using a SQL parser and random number to accept the SQL keyword according to that random instruction set. Thus, any SQL keyword without the number or out of the range will be rejected (Boyd & Keromytis, 2004). Static Analysis approaches, by analysing web application to detect the vulnerable SQL query sentence in addition to validate the user input. Gould et al use JDBC checking tool to check statically for type correct queries in the SQL statement that are generated dynamically in Java. This technique is not effective enough to detect all type of SQL injection vulnerabilities because it was not developed for this purpose or not for prevention attack, but it is usable. (Gould et al, 2004). X et al propose SAFELI tool which is a white box analysis tool that analyses an asp.net application. It depends on the analysis of the byte code of the application, and previous collected information about attack patterns. This collected information is used during the analysis as attacks behaviour reference. In addition, SAFELI analyses the web application using the symbolic execution engine which clarifies all the application pages with it entry points (Fu et al, 2007). Static and Dynamic Approaches, Halfond and Orso use static analysis procedure to build the SQL query model that determines the construct queries points which have direct access to the database. Successively, each of the construct queries points will be supported by runtime monitoring that investigates the queries before sending it to database. This investigation checks those queries against any existing attack. However, the limitation is the monitoring step that depends on the result of static analysis step, and that means if there is a fault in the first step then the fault will be in the other step (Halfond et al, 2005). Ruse and his colleagues suggest another solution against SQL injection attacks by developing an automatic model to capture any change of the sub-queries and its dependencies using CREST (test tool for C programs). The automatic model runs through three main steps starting from compiling SQL statement to be usable with c programs. The compiled sentences will be tested by using CREST which is generate the test to check the injection possibility by detecting the injection causing requirements, and at the runtime the user input will be monitored to find any of requirements that is detected in the previous step. The feature of this approach is there are no false positives like in other static analysis approaches, and it also looks at the semantic structure of the SQL query and not on the syntactic structure like other previous approaches. (Ruse et al, 2010). Lee et al use the combination of static and dynamic technique by removing any of SQL attribute value of the SQL query at runtime and compared it with a static SQL query (Lee, 2011).
All the mentioned techniques are looking for the SQL injection attacks as one step or as a static attack but SQL attacks are dynamic or done in several steps which is the main assumption of our approach.

3. Detection and Prevention Framework DPF

3.1 Overview of DPF

DPF is used to monitor and block SQL injection attacks and it uses the Anatempura tool as a monitoring tool to block malicious users. DPF is initialized by specifying some of the existing attack patterns using ITL, and connecting the Anatempura tool to the web server to monitor the user input data. Broadly, DPF is divided into three main phases: the initial phase, the checking phase and the decision phase as show in Figure 1. The initial phase consists of several steps starting from the user input till the data arrives to the input checker which is the first step of the checking phase. In the checking phase, the system will analyse the data using the input checker which uses Anatempura to analyse the user entry against existing attack patterns, and thus to decide whether it is a bad or good input. The checking phase consists of three processes which are input checker, output checker and database observer and the result of the checking phase will be sent to the decision phase. The decision phase can be divided into two main parts which are the feedback and user behaviour components. The three phases will be explained in more below.

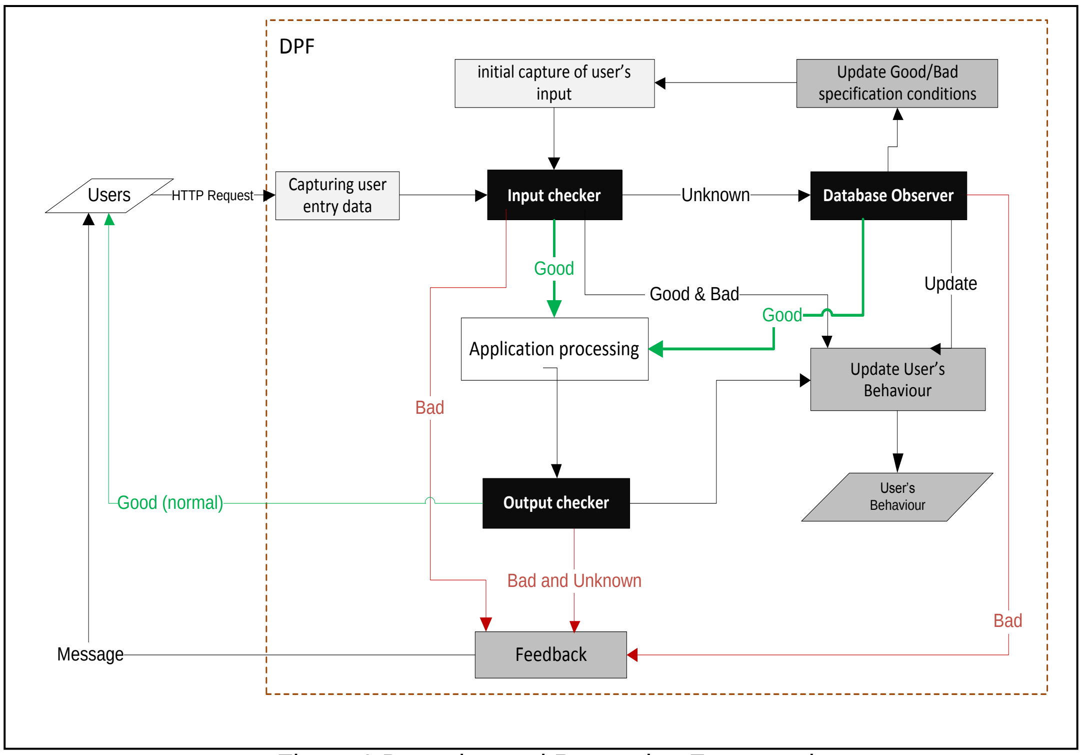

Figure 1 Detection and Prevention Framework

3.2 Description of each component

This section describes the components of our detection and prevention framework (DPF).

Initial capture of user input, it is considered the basis of DPF to bootstrap the system by specifying the good and bad input at the character level using ITL. The good input should not includes any symbol like single quotation and double quotation or star, or in other word the good input should not contains any of SQL keywords that can be used to attack the web application database. Furthermore, the bad input will be specified using ITL by specifying some of the existing attacks patterns (SQLlib, 2007). Note, the initial bad / good specifications that are specified using ITL will be used by the Anatempura tool.

Users, the users are considered as a part of the initial phase because any transaction could be started from the user. Therefore, the user is anyone who performs a transaction on the system database.

Capturing Data will analyse the http request to extract the user entry data and convert it to Anatempura format (variable, value, timestamp). This component does not depend on Anatempura but on the web page application type. In other word, the data will be extracted using library calls offered by the programming language that is used to develop the web application.

Input checker will use the existing attack patterns that are prepared by the initial capture of user input component. The checker will analyse the user entry against the existing attack patterns, and the outcome is as follows:
The entry data is good, so the data will be passed to the application server for normal processing, and the user's behaviour will be updated accordingly. If the entry data type is a bad, then the data will be rejected and therefore DPF updates the user's behaviour and prepares to send a message to the user via the feedback component. The last possibility is that the entry data is unknown, then the database observer will detect the bad/good entry according to the observations of the observer. The following functions have been created to check the input:

Input Seq.	Function Name	Function Aims
1	DecreaseSpaces()	To remove any extra spaces in the user input
2	LowerCase()	To transform all of the user input to lower case
3	Goodentry() and SearchTokens()	Check of the user input whether it includes of SQL keywords
4	Badentry()	To check the input whether it includes any of the existing attack pattern
5	BehaviourDetection()	To model user behaviour

Table 1 Checking Functions

All of the mentioned functions in Table 1 are combined in one procedure which is the CheckingModel.

Database Observer will check unknown entry cases which are not caught by the initial analysis of user entry data via the input checker. The main idea of the DB observer is to determine what will happen in the DB engine, and this will be done by monitoring each transaction that comes from the input checker as unknown entry. Moreover, the checking of the transaction depends on the developer because the DB observer needs the developer to specify the expected result of each transaction that is run by the developed web application such as, the table name, running command type, number of the expected records, and the user type. The expected result will be compared with the runtime result. Furthermore, the comparison between the expected result and the runtime result is used to ensure that the database transaction was performed safely (if the runtime result is similar to what the developer expected) as shown in the Figure 2.

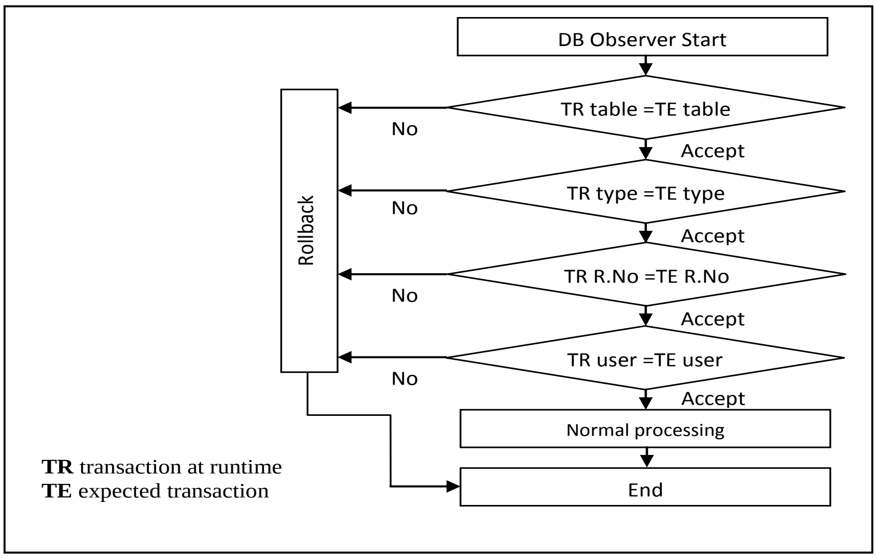

Figure 2 Database Observer

Therefore, the database observer has to monitor four conditions that are specified by the developer or the programmer as follows:

- ➢ Transaction type of the real execution is the same as the expected. For example, if the transaction type at the real execution is “Select” and the expected one is “Select” as well, then the database observer will accept this transaction at this step and continue. If they are different the database observer will do a rollback and respond to the input checker to reject the entry data and update the user behaviour.
- ➢ The table name of the real execution is the same as the expected one. For example, if the transaction table at the real execution is “users” and the expected table is also “users” the database observer will accept it and continue to the next step. If they are different then the database observer will do a rollback and this transaction will be rejected.
- ➢ The record number of the real execution is the same as the expected number. For example, the login page normally returns one record with the select statement, so if the real execution of the select statement returns the same number (one record), then the database observer will accept it, otherwise it will be rejected and the database observer will do a rollback.

➢ User type of the real execution is the same as the expected one. For example, if the user tries to change the password at the change password page therefore the user type should be same as expected one. However, if the user tries to change another user's password then the database observer will catch this and this transaction will be rejected and the database engine will do a rollback. Note the database observer can only deal with recoverable transactions so no DDL (Data Definition Languages) commands like create, drop, and alter table, because the DDL injected command cannot be recovered by rollback command, so these transactions should be rejected before accessing the database by the input checker.

Output checker will check whether the message sent to the user is safe or not. Moreover, the output checker will not analyse the response in the same way as the input checker, because it will block any message that contain any detail about database structure or type because these type of messages are not safe. Moreover, the output checker will block the unsafe messages using the library calls in the programming language which is employed to develop the web application.

Feedback component prepares the message that will be sent to the user regarding the cases of bad entry data. Moreover, if the user enters the data using a bad method and the input is caught as unsafe then DPF will respond to this entry by using prepared messages relevant to the entry method.

User's behaviour will track each transaction in the system and detects the type of the transaction, i.e. whether it is a good or bad transaction. The tracking information will be used to model the user behaviour. Therefore, the user behaviour depends on the result of the input and output checker in addition to the result of the database observer. The DPF will use user information like IP address, user status (good, bad), attacking technique (primitive, advance) , and time stamp of the transaction to build the user behaviour as shown in Figure 3.

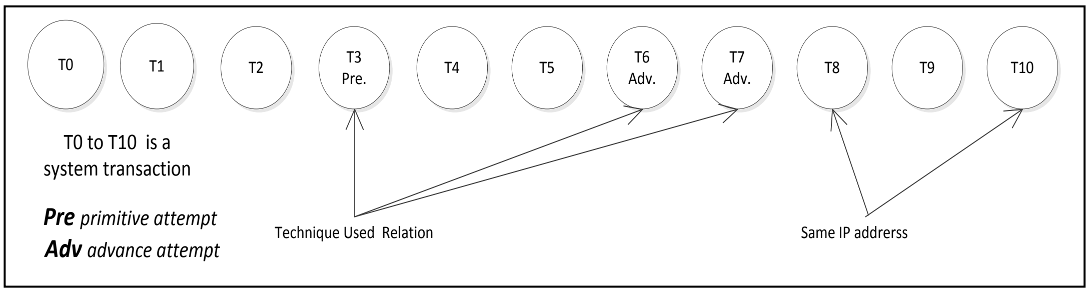

Figure 3 Transactions Relation

Moreover, the DPF can determine user behaviour of three types i.e. normal, good and bad according to three criteria's namely, the percentage of transaction, the sequence of the same transaction type, the transaction types as shown in Figure 4.

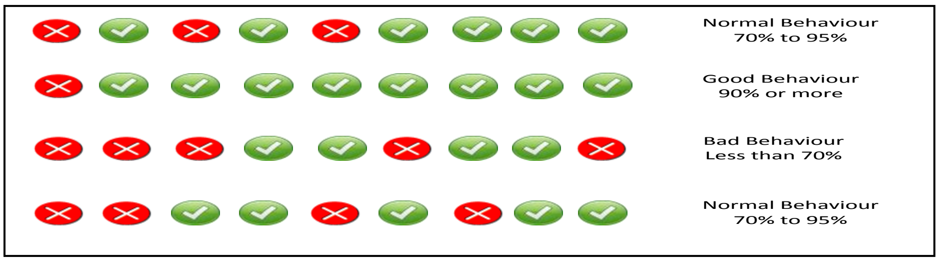
Normal Behaviour 70% to 95%
Good Behaviour 90% or more
Bad Behaviour Less than 70%
Normal Behaviour 70% to 95%

Figure 4 User Behaviour

Figure 4 shows user behaviour of various types: good behaviour if there are 95% or more of good transactions, the bad behaviour if less than 70% of good transactions and any other behaviour will be considered normal behaviour (between 70% to 95%). Additionally, user behaviour will also be classified as bad behaviour if there is a sequence of three or more bad transactions. Therefore, user behaviour can be used as a quick view of the transactions status and the proportion of hacking attempts that have been done so far. It can act as an early warning to the system.

DPF Updates

In DPF there are two types of updates which depend on the input data as follows:

Updating of User's Behaviour will update continuously the user behaviour within DPF according to the checking result of the checking processes the input checker, database observer and output checker.

Updating of Existing attack patterns this component is working in parallel with the database observer component, because when the database observer finds any unsafe inputs, it will send this information to this component to update the existing attack patterns. Note, the updating of the attack patterns library will be done manually, because the library that is used by input checker is specified in ITL, thus the updating should use some translation into ITL to be usable with Anatempura.

4. Anatempura

Anatempura is a tool for the runtime verification of execution interval temporal logic formula. Interval Temporal Logic (ITL) is a flexible notation for both propositional and first-order reasoning about periods of time found in descriptions of hardware and software systems. Unlike most temporal logics, ITL can handle both sequential and parallel composition and offers powerful and extensible specification and proof techniques for reasoning about properties involving safety, liveness and projected time. Timing constraints are expressible and furthermore most imperative programming constructs can be viewed as formulas in a slightly modified version of ITL. Tempura provides an executable framework for developing and experimenting with suitable ITL specifications. In addition, ITL and its mature executable subset Tempura have been extensively used to specify the properties of real-time systems where the primitive circuits can directly be represented by a set of simple temporal formulae. In addition, various researchers have applied Tempura to hardware simulation and other areas where timing is important. [1]

Anatempura, which is built upon C-Tempura, is a tool for the runtime verification of systems using Interval Temporal Logic (ITL) and its executable subset Tempura. The runtime verification technique uses assertion points to check whether a system satisfies timing, safety or security properties expressed in ITL. The assertion points are inserted in the source code of the system and will generate a sequence of information (system states), like values of variables and timestamps of value change, while the system is running. Since an ITL property corresponds to a set of sequences of states (intervals), runtime verification is just checking whether the sequence generated by the system is a member of the set of sequences corresponding to the property we want to check. The Tempura interpreter is used to do this membership test.[2]

5. Case study

We discuss a case study to clarify how attacker's information can be used to model user behaviour. This case study describes an input scenario and assumes the status of each input is already determined by Anatempura via the CheckingModel procedure as mentioned in section 3. The table below shows a particular input scenario.

Input Seq.	User IP	The input	status
1	146.168.255.12	Normal	g
2	146.168.255.13	Normal	g
3	82.164.254.12	' or '1'='1	b
4	82.164.254.12	';drop table users;--	b
5	212.164.254.14	Normal	g
6	212.164.254.16	Normal	g
7	212.164.254.14	any'; declare @NewStoreProcedure char(80) …….;	g
8	67.164.254.14	normal	g
9	146.164.2.46	' ;	b
10	212.164.254.14	normal	g
11	182.164.254.23	any'; EXEC (@NewStoreProcedure);	b
12	212.164.254.14	Normal	g
13	212.164.254.16	Normal	g

Table 2 Selective User's Inputs

[1] ITL Home Page http://www.cse.dmu.ac.uk/STRL/ITL/
2 Anatempura http://www.cse.dmu.ac.uk/STRL/ITL/itlhomepagese6.html#x7-70006.1

The input status column lists the status of each input. The following ITL determines whether a particular scenario is named or not:

$\Diamond$ (Bad $\wedge$ IP =IP_0) : $\Diamond$ (bad $\wedge$ IP =IP_1) $\Rightarrow \mathrm{IP}_0 = \mathrm{IP}_1$

This means if an IP in a certain state is equivalent to an IP in a previous state then these state are related.

$\Diamond$ declare command: $\Diamond$ EXEC command $\Rightarrow$ true: declare command: true: EXEC command $\Rightarrow$ skip; ($\Box$ EXEC command) ; skip

This means If there is an exist executing of command in a certain state and the declaration of the same command in previous state then these state are related.

In Table 2, input 3 and 4 are marked as bad, those attempts are one step attacks because they do not retrieve any information from the database and just try to inject the harmful code in the web application fields. However, those attempts have the same IP address which means there is a relation between them because both attempts have been done by the same user. The input 7 and 11 can be classified as related as well, because the attacker here tried to declare the stored procedure in the first attempt and in the second attempt he / she use it. So there is a relation between these hacking attempts and this justifies the use of monitoring user behaviour.

6. Conclusion and future work.

This paper has presented a new approach to detect and prevent SQL injection attacks. The paper highlighted some of the previous existing techniques used against these types of attacks. Our technique is classified as a runtime detection and prevention technique. It uses the Anatempura tool to specify existing attacks patterns and to model user behaviour. We illustrated our approach through a case study that describes a scenario where several input are related in any an ongoing attack. Those relations can be have been defined in ITL to successfully model user behaviour. The paper shows our framework in detail by clarifying the detection processes how those process works with the other components in the framework. Finally, our approach is still under development and we will enhance its functionality by running more experiments that model user behaviour. Future work will consist of completing the implementation to evaluate it will real world SQL injection attacks scenario.

References:

[1]. Gould, C., Zhendong, Su. & Devanbu, P. (2004), ‘JDBC Checker: A Static Analysis Tool for SQL/JDBC Applications’, in 'ICSE', IEEE Computer Society, pp. 697-698.

[2]. Halfond, W. G. J. & Orso, A. (2005), ‘AMNESIA: analysis and monitoring for NEutralizing SQL-injection attacks’, in David F. Redmiles; Thomas Ellman & Andrea Zisman, ed., 'ASE' , ACM, pp. 174-183 .

[3]. 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 (1-2), 58-68.

[4]. SQLlib-tool (2007), Open labs web application security. http://www.open-labs.org/sqlibf113b2.tar.gz Accessed [12-10-2011].

[5]. OWASP Top 10 for 2010. https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project Accessed [09-09-2011].

[6]. Ruse, M., Sarkar, T. & Basu, S. (2010), ‘Analysis & Detection of SQL Injection Vulnerabilities via Automatic Test Case Generation of Programs’, in ‘SAINT’, IEEE Computer Society, pp. 31-37.

[7]. Kosuga, Y., Kono, K., Hanaoka, M., Hishiyama, M. & Takahama, Y. (2007), ‘Sania: Syntactic and Semantic Analysis for Automated Testing against SQL Injection’, in 'ACSAC' , IEEE Computer Society, pp. 107-117.

[8]. Fu, X. et al (2007), ‘A Static Analysis Framework for Detecting SQL Injection Vulnerabilities’, in Proceedings of 31st Annual International Computer Software and Applications Conference (COMPSAC 2007), pages 87 – 96.

[9]. Fu, X. & Qian, K. (2008), ‘SAFELI: SQL injection scanner using symbolic execution’. in Tevfik Bultan & Tao Xie, ed., 'TAV-WEB' , ACM, , pp. 34-39.