Comparing SQL Injection Detection Tools Using Attack Injection: An Experimental Study

Ivano Alessandro Elia
Department for Technologies,
University of Naples *'Parthenope'*
Naples, Italy
ivano.elia@uniparthenope.it

José Fonseca, Marco Vieira,
CISUC, Department of Informatics Engineering (DEI),
University of Coimbra
Coimbra, Portugal
josefonseca@ipg.pt, mvieira@dei.uc.pt

Abstract— **System administrators frequently rely on intrusion detection tools to protect their systems against SQL Injection, one of the most dangerous security threats in database-centric web applications. However, the real effectiveness of those tools is usually unknown, which may lead administrators to put an unjustifiable level of trust in the tools they use. In this paper we present an experimental evaluation of the effectiveness of five SQL Injection detection tools that operate at different system levels: Application, Database and Network. To test the tools in a realistic scenario, Vulnerability and Attack Injection is applied in a setup based on three web applications of different sizes and complexities. Results show that the assessed tools have a very low effectiveness and only perform well under specific circumstances, which highlight the limitations of current intrusion detection tools in detecting SQL Injection attacks. Based on experimental observations we underline the strengths and weaknesses of the tools assessed.**

Keywords- Security; SQL Injection; Web applications; Intrusion Detection; Fault Injection

I. INTRODUCTION

Web applications are the backbone of today's business and are enormously widespread. Nearly all information systems and business applications (e-commerce, banking, transportation, web mail, blogs, etc) are now available as web-based database applications. The demand for web applications also attracts adversaries that want to benefit from their weaknesses. A survey conducted in 2007 estimated that 70% of the web applications are at risk of being hacked [1] and the fact that these applications can be accessed from everywhere in the globe, makes them even more interesting for attackers .

One of the most used techniques to attack web applications is SQL Injection [2, 3]. SQL Injection takes advantage of unchecked input fields at the web application interface to maliciously tweak the query sent to the back-end database. Through this attack, adversaries can access and manipulate database data that should only be handled by a restricted set of people [4].

The database is the key element of most web applications as it is typically where the most important assets are stored. This way, a security breach exposing the database worldwide can be fatal to the enterprise reputation and liability. This also affects clients and personal data, exposing them to unforeseen risks. SQL Injection is very profitable for attackers and there is a flourishing black market that trades all sort of digitally stolen goods, like credit card numbers and bank accounts [5]. Most of the time attackers can even escape without being prosecuted due to bad server or application configurations, lack of logs (or the ability to analyze them), and lack of security regulations from corporations and governments [5].

To address the problem of lack of security in web applications some preventive measures must be taken, like code reviewing and penetration testing [6, 7]. However, the incredible growth of web applications both in number and in complexity poses unavoidable restrictions to what can realistically be done to protect them [4]. Human resources are limited and, in many situations, they are no longer able to deal with the huge amount of source code present in web applications within an organization (and with their ramifications to other systems). Furthermore, economic constraints restrain the transfer of resources to the security department in spite of the increasing number and complexity of security procedures.

The use of automated security tools is mandatory nowadays. In fact, for the security community it is important to build tools that allow an easier evaluation and prevention of security problems. Furthermore, the demand for security tools from corporations and government agencies is increasing. A survey done by the Computer Security Institute/FBI with responses from over 500 computer security practitioners concludes that 55% of respondents use automated tools to evaluate security [8]. However, the widespread use of these tools grows far faster than the ability to test and verify their real effectiveness in concrete scenarios. Obviously, SQL Injection detection tools are no exception and, although they are key elements for guaranteeing the security of web applications, their users need procedures to assess them systematically [9]. In practice, their effectiveness needs to be carefully evaluated, and this represents a major concern among security practitioners [9].

A key problem is that tool vendors typically advertise good quality and good performance, but final users do not know how and in what conditions were the tools evaluated. It is also known that security tools do not provide a complete coverage of all the problems (i.e., no tool solves all the

problems). According to a US NSA Center for Assured Software evaluation procedure, cited by a Softtek paper, it was observed that a single tool could only detect 60% of the application vulnerabilities [10]. However, in spite of the importance of tool assessments, there are no structured ways to evaluate security related tools in custom environments.

In this paper we assess the effectiveness of five intrusion detection tools for SQL Injection in custom setup scenarios: Apache Scalp, ACD Monitor, GreenSQL, Snort and a DB IDS. Our methodology applies to the security area of the fault injection technique used for decades in the assessment of fault tolerant systems [11]. To analyze key figures of merit of the tools being evaluated, we inject realistic vulnerabilities in custom deployed web applications and then automatically attack those vulnerabilities, using a state of the art Vulnerability & Attack Injector tool [12]. The security tools under assessment are then monitored in order to observe the detection coverage of the attacks. Obviously, we expect that the tools can detect the attacks in due time without a significant number of false positive alarms, however, our findings clearly show that this is not usually the case. In fact, the low coverage observed in the experiments highlight many limitations of these types of tools.

The structure of the paper is as follows. Section II presents the evaluation methodology and tools and Section III describes the experimental setup and procedure. Section IV presents and discusses the results. Finally, Section V concludes the paper and introduces future work.

II. Evaluation Methodology and Tools

In this section we overview the evaluation methodology, namely the attack injection approach, the web applications considered, and the SQL Injection detection tools assessed.

A. Attack Injection

The Attack Injection methodology consists of injecting realistic vulnerabilities in a web application and performing controlled attacks [12]. It can be seen as the translation of the traditional fault injection technique to the web application security scenario. The underlying idea is to explore the web application code and to inject vulnerabilities so that they look like real vulnerabilities, both in their type and in their location. The vulnerabilities injected are automatically attacked and the results analyzed (Fig. 1).

One of the main goals of this methodology is to test web security mechanisms by providing a realistic test bed based on ad-hoc web applications. Security mechanisms can be stressed realistically in this custom scenario. This allows comparing several tools in specific environments, which is difficult to achieve by other means. For example, a security practitioner can use this Attack Injection procedure to select the intrusion detection tool (or combination of tools) that offers better results in a particular web application setup. This methodology has three stages that are executed sequentially:

- **The Preparation stage** – In this stage the web application functionalities are interacted while both HTTP and SQL communications are captured and processed.
- **The Vulnerability Injection stage** – The web application code is analyzed using a vulnerability injection methodology, resulting in a collection of web application source code files, each one having a single vulnerability injected.
- **The Attack stage** – A new interaction with the web application is performed. All vulnerabilities generated are injected, one at a time. A collection of attack payloads is applied to try to exploit the vulnerabilities injected. The process is repeated until all the vulnerabilities have been injected and attacked. Information about the attack success data and metadata is delivered as output.

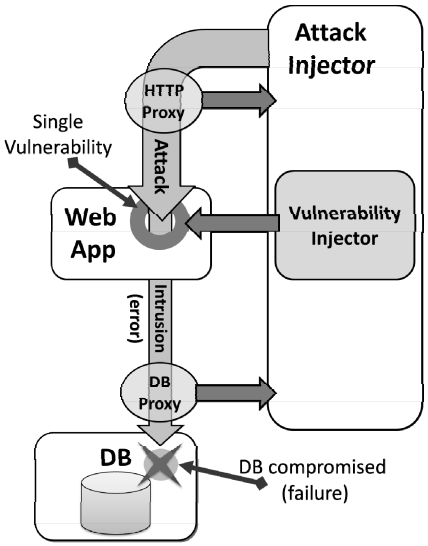


Figure 1. Overview of the Attack Injection methodology.

In the following subsections we introduce these stages, applied to a common setup consisting of a LAMP (Linux, Apache, MySQL, PHP) web application supported by a MySQL back-end database in a Ubuntu Linux OS. The architectural layout of the Attack Injection Tool is shown in Fig. 2 (for details on the Vulnerability & Attack Injection methodology see [12]).

1) Preparation stage

The objective of this stage is to obtain dynamic information about the web application pages, their GET and POST variables, some correct values of these variables and how these variables are used inside the queries sent to the database.

This stage starts with the deployment of two proxies: the HTTP proxy and the Database proxy. These proxies are used to monitor all the HTTP and database communications and send a copy back to the Attack Injector. The next step is to crawl the web application. This can be done automatically by filling up all the form fields and following every hyperlink in order to discover all the web application pages.

The HTTP proxy collects the code of the web application pages sent to the browser, their GET and POST variables and respective values. When, as a result of the web interaction, a

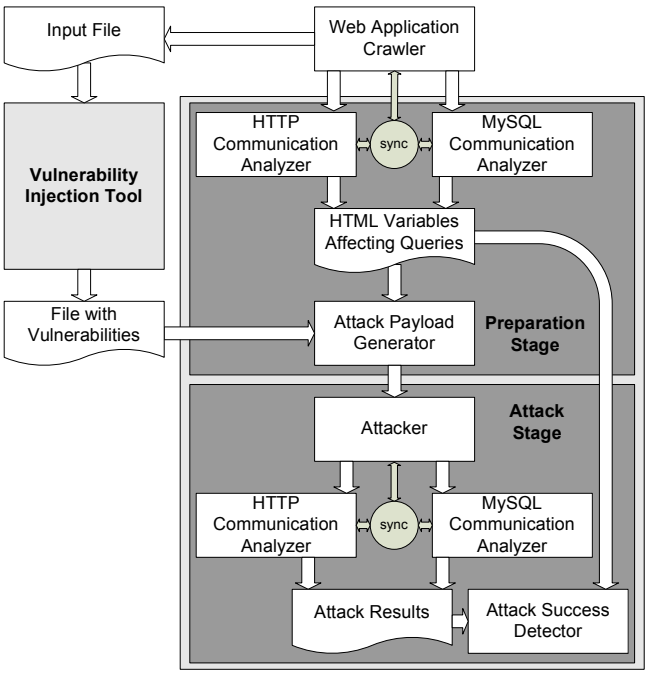


Figure 2. Architecture of the Attack Injection tool [12].

query is submitted to the database, the Database proxy captures it together with the response of the database. For this stage to be successful, the input interaction monitored by the HTTP proxy is synchronized with the Database communication proxy. This allows understanding the input variables that affect the SQL queries and how. The resulting data is used in the next stages.

2) Vulnerability Injection stage

In the Vulnerability Injection stage all the web application pages discovered in the Preparation stage are statically analyzed. The variables obtained by this static analysis are correlated with those that came from the dynamic analysis performed in the Preparation stage. The goal is to gather information about the input variables that are used in SQL queries and, therefore, could be the entry point to attack the web application (if they were vulnerable to SQL Injection).

The first step to artificially inject SQL Injection vulnerabilities is to know the characteristics of the web application variables that affect SQL queries. This is performed by the Vulnerability Injector component (Fig. 1), which is based on the Location Pattern and Vulnerability Code Change attributes of real world SQL Injection bugs. The **Location Pattern** defines where a vulnerability may be injected and the **Vulnerability Code Change** defines what has to be changed in the code to inject a vulnerability. For every location in the source code where a given variable is likely to be found unprotected (Location Pattern), the Vulnerability Injector component tweaks the code so that a realistic vulnerability is injected there (Vulnerability Code Change). This change in the code, that tries to break the variable's protection, is temporally stored in a separate file and will be used during the Attack stage to apply the vulnerability prior to the attack.

In the Vulnerability Injection stage a collection of attack payloads is also generated for each vulnerability. This will be used during the Attack stage to exploit the vulnerability. For each attack payload a footprint is also created, which will be used in the next stage to assess the success of the attack.

3) Attack stage

In this final stage, the vulnerable files that resulted from the previous stage are injected in the web application, the attack is executed and the results are analyzed. For the attack procedure, once again, the HTTP and Database proxies are deployed. However, now they are used to monitor and assess the success of the attack. Multiple attacks are performed in this stage and, because of that, the web application files and the database data are backed up beforehand.

The attack procedure starts by injecting a single vulnerability in the web application code. This is done by overwriting the original web application file with one of the corresponding copies provided by the previous stage (each copy has only one vulnerability injected and they are all different from each other). Afterwards, the vulnerability is attacked by executing one of the attack payloads generated for that vulnerability. The results of the attack gathered by the HTPP and Database proxies are stored for posterior analysis.

Before the next attack, the database is restored from the backup copy. Then the next payload is applied and the process is repeated until no more payloads exist for the current vulnerability. After finishing all the attacks prepared for the vulnerability injected, both the web application files and the database are restored from the backup copy. The next vulnerability is injected and the procedure executed from the start and repeated until all vulnerabilities have been attacked and the results collected.

The final action of the Attack stage is the analysis of the resulting data to decide which attacks were successful (or not). This decision is based on searching the corresponding payload footprint in the outcome of each attack. For an attack to be considered successful, the payload footprint has to be found in the communication data captured by the HTTP and Database proxies.

B. Web Applications

The key idea in this experimental study is to artificially inject and attack SQL Injection vulnerabilities in web applications, while monitoring the behavior of attack detection tools. Obviously, this process of code injection restricts the web application target to those that have their source code available. However, this type of web applications is quite common and it is widely adopted around the globe, representing a large slice of the web application market. In particular, open source LAMP web applications are one of the most common worldwide [13][14], and that is the key reason why we have chosen to use them in our work.

LAMP stands for a widespread combination of technologies including a Linux OS running the Apache web server and a Mysql database that supports a PHP developed web application. All the components of the LAMP environment are open-source, cheap to deploy and easy to work with, making them truly engaging for developers,

including those with little experience. Obviously, all these attributes attract many developers to the world of web applications, even those with little or total absence of security knowledge. Unfortunately, this same lack of security concern can also be observed in the administrator side of LAMP. As a result, we have lots of web applications that have a huge number of people using them, but that are completely vulnerable to attacks.

To simulate the attack scenario, in our experiments we used three web applications: TikiWiki [15], phpBB [16], and MyReferences. They represent setups of different complexity, including two well known and award winning web applications (TikiWiki and phpBB) and one custom-made application (MyReferences):

- **TikiWiki** is a groupware/content management system that permits building a wiki, which is a web site allowing users to contribute to it by adding and modifying the contents of web pages. It is widely used for building sites, such as the Official Firefox Support site and the KDE wiki. It was one of the finalists of the sourceforge.net 2007 for the most collaborative project award.
- **phpBB** is a well-known web application and it has become the most widely used open-source forum solution. It is used by millions of users worldwide and it was the winner of the sourceforge.net 2007 community choice awards for best project for communications. It is also the forum module integrated into the popular phpNuke content management and portal web application.
- **MyReferences** is a custom-made web application used to manage publications and bibliographic references. It allows the storage of the source of the documents (usually a PDF file) and some metadata information such as the title, the publisher, the year of publication, the document type, the relevance, and the authors. The information may be edited, queried and displayed. It is a small application (it consists of only 13 PHP files) that represents simple home made web applications that are extremely common due to the ease of web development.

C. SQL Injection Detection Tools

The set of tools considered in this study comprises both well-known security tools and innovative instruments from the research field. It includes heterogeneous types of security tools that address different strategies for SQL Injection detection, including log analyzers, network sniffers, and proxies. The approach used for the detection also differs, as some tools are based on anomaly-detection while others are signature-based [17]. Moreover, all the different architectural levels (i.e., Network, Application and Database) are monitored and analyzed by different tools, which provides a wider range of analysis. The tools tested in this work are (see summary in Table I):

- **Apache Scalp** (version 0.4) [18] is an analyzer of the Apache web server access log. Using the log data, Scalp can detect several types of web application-related attacks, by comparing them with a set of known signatures for the most important web application attacks, like SQL Injection, Cross Site Scripting, Cross Site Request Forgery, Path Traversal, among others. Detection is achieved by analyzing the HTTP request sent by the client web browser to the apache web server, which is stored in the access log.
- **Anomalous Character Distribution (ACD) monitor** is a detection tool that works at the application-level and uses an anomaly-based approach (similar to the one described in [19]). Like Scalp, ACD looks for HTTP requests in the web server access log and, for each request, it associates an anomaly score based on the character distribution of the requested URI. Before using the ACD monitor in detection mode, it must be trained with a clean access log, composed only of legitimate requests. A configurable threshold value indicates the minimum deviation from the legitimate profile distribution to be considered malicious.
- **GreenSQL** (version 1.2.2) [20] is an open source database tool specifically targeted to protect MySQL databases against SQL Injection attacks. GreenSQL is deployed as a proxy that intercepts all the SQL communication with the database. For each SQL query it builds a risk-scoring matrix that indicates the probability of the query to be malicious. It can also work as a database firewall, preventing supposed malicious queries from being performed or as an Intrusion Detection System (IDS) that alerts the administrator when such queries are executed.
- **Snort** (version 2.8.4.1) [21] claims to be "the most widely deployed (network level) IDS/IPS technology worldwide" [22]. Snort is based on a set of rules (rules version 2.8) that are used for detecting web application attacks. However, most of the rules are signatures of attacks to very specific web application vulnerabilities. For this reason in our experiments we introduced a set of more generic rules for the detection of SQL Injections, as proposed in [23].
- **DB IDS** [24] is a database level IDS for Oracle and MySQL databases. This IDS implements an anomaly detection approach based on a training phase and a detection phase. Before detection, the IDS must be trained for the target web application. This training can be done by interacting manually with the web application, but to be systematic, a web crawler that executes all the web application functions can be used (this was the chosen option in our experimental evaluation).

TABLE I. TOOLS CHARACTERISTICS

Tool	Architectural Level monitored	Detection Approach
Apache Scalp	Application	Signature Based
Anomalous Character Distribution monitor	Application	Anomaly Based
GreenSQL	Database	Signature Based
Snort	Network	Signature Based
DB IDS	Database	Anomaly Based

III. SETUP AND PROCEDURE

In this section we describe the evaluation setup and procedure, focusing particularly in the testbed, evaluation steps, and metrics.

A. Experimental Testbed

A schematic representation of the testbed that was prepared to conduct our experiments is presented in Fig. 3. The testbed is based on an Ubuntu Linux box where the Attack Injector was deployed along with an Apache web server and a MySQL database. The Attack Injector is also represented with its HTTP and MySQL proxy, which are needed to analyze the web application and monitor the attacks.

The tools were deployed in this environment in the least intrusive way, avoiding interfering with the operation of each other and with the Attack Injector. In order to do so, the following configuration policies were applied:

- Apache Scalp and the ACD monitor only need to monitor the Apache's access log so they are transparent to the Attack Injector and to the other tools, thus their deployment is straightforward.
- As DB IDS and Snort use network sniffers as data sources, to avoid detections over duplicated data flows due to the use of multiple proxies. Tools were configured to analyze only a single HTTP flow and a single MySQL flow (the ones effectively used by the web applications).
- As the Attack Injector was placed in the same machine of the web server, Snort detects the interaction in the loopback interface, therefore triggering Snort's alerts for loopback and same source/destination traffic. This prevents all the other rules from being evaluated (which are precisely the ones that we want to evaluate). To overcome this problem, the HTTP traffic was redirected from the loopback interface to a real network interface using an additional proxy. This is not shown in Fig. 3, as it would make the representation unwieldy and, in practice, it does not affect the setup design logic.
- GreenSQL had to be deployed as an additional proxy. We placed it between the Attack Injector's MySQL proxy and the real database in order to limit the changes in the setup. In this way we had to change the configuration of the MySQL proxy, but the configuration of the web application and of the other probes remained untouched.

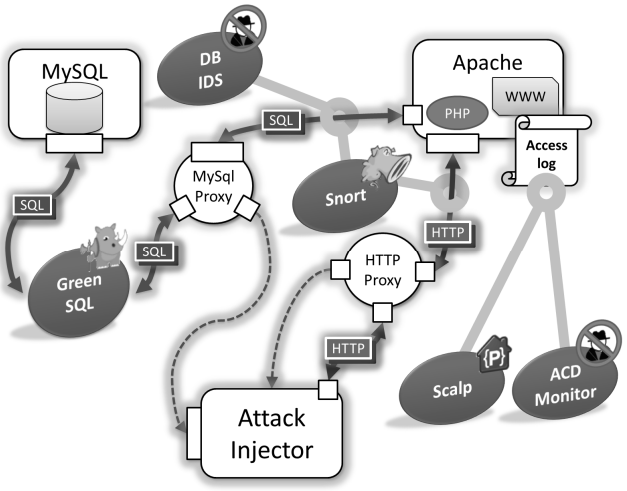


Figure 3. Schematic reperesentation of the testbed.

B. Evaluation Procedure

After the deployment and configuration of the tools the setup is ready to start the experiments. Due to the differences in the way the tools collect the data and their specific needs and constraints, each one is tuned in its own way during the three stages of the Attack Injector workflow.

The procedures executed during the **Preparation Stage** of the Attack Injection are also useful to automatically configure the setup needed to correctly tune GreenSQL, DB IDS and ACD tools for the specific web application. This includes training of the tools, which requires a learning phase to work correctly. Two important aspects are:

- During crawling all the tools are switched off in order to avoid unwanted alerts, except the GreenSQL and the DB IDS that, in this stage, are set in the "learning mode". During the learning mode GreenSQL creates a white list of queries and the DB IDS learns the characteristics associated with the normal operation of the web application.
- During crawling the Apache web server populates the access log with specific HTTP request data. When the Preparation Stage finishes, this data is used to train the ACD monitor with the character distribution of the legitimate requests.

The preparation phase is also the moment for establishing the lowest acceptable threshold value to be used with the ACD monitor for each web application. This value may be different for each web application and must guarantee that the normal interaction with each one does not produce too many alerts. For this, we tested the tool (previously trained for each specific web application) using different values for the threshold. We identified the lowest acceptable value for the threshold as being the one that produced less the 1% of false positives detections during the crawling (the different thresholds used are discussed later in Section IV).

During the **Vulnerability Injection Stage** no operation is needed for the detection tools, so they are disconnected. For each web application under test, the Attack Injector injects a set of vulnerabilities and for each vulnerability injected the Attack Injector is configured to generate seven representative types of malicious payloads (Table II).

The detection tools are switched on during the **Attack Stage** of the Attack Injection, according to their needs:

- GreenSQL is configured to operate in IDS mode, alerting for suspicious queries, but not stopping any operation on the database, as it would be the case in the firewall or IPS mode. This way it is working in the least intrusive manner (without loosing any detection capabilities) so it will not affect the Attack Injector analysis of the attack results.

TABLE II. ATTACK INJECTOR'S PAYLOADS

Attack Payload Strings	Expected result of the attack
‘	Change in the structure of the query. The query result is an error
or 1=1	Change in the structure of the query. The query result is the override of the query
‘ or ‘a’=’a	Change in the structure of the query. The query result is the override of the query
+connection_id() -connection_id()	Change in the query. The query result is 0
+1-1	Change in the query. The query result is 0
+67-ASCII('A')	Change in the query. The query result is 0
+51-ASCII(1)	Change in the query. The query result is 0

- Since the detection performance of the ACD monitor is strongly influenced by the threshold used to identify the malicious requests, five instances of this tool where used during the experiments, each one using a different value for the threshold (see details in Fig. 4, Fig. 5 and Fig. 6).
- Finally, as aforementioned, some rules for snort where added to the default ones in order to detect generic SQL injection attempts not associated with known vulnerabilities [23].

After completing the attack stage (that includes identifying the attack success for each of the payloads injected), all the alerts generated by each tool were collected and evaluated against the attacks (both successful and unsuccessful) reported by the Injector. Since the output of every tool has a different format, we used a set of adaptable parsers and scripts in order to render the alerts in a common format and make the evaluation more systematic.

C. *Performance Evaluation and Analysis*

To characterize the effectiveness of the tools we focus on the coverage metric. Note that coverage targets the percentage of attacks that are detected, but we should keep in mind that not all the attacks resulted into the successful injection of a malicious payload in the database (which is the final goal of SQL Injection). For this reason, we take into account the evaluation made by the Attack Injector about the successfulness of each attack attempt, and actually analyze three key metrics:

- **Percentage of attack attempts detected** – percentage of attacks detected by the tool comparing to the total number of attack attempts performed by the Attack Injector (including both successful and not successful).
- **Percentage of successful attack attempts detected** – percentage of attacks detected by the tool comparing to the attack attempts that were reported as successful by the attack injector (i.e., SQL Injection attacks that reached the database server).
- **Percentage of unsuccessful attack attempts detected as attacks** – percentage of attacks reported by the tool comparing to the attack attempts that were reported as not successful by the attack injector (i.e., SQL Injection attacks that did not reach the database server).

Obviously, not all the three metrics are equally relevant with respect to all the tools under test. For instance, for the tools that are intrinsically not capable of differentiating successful attacks from the not successful ones (e.g., tools that analyze the web server logs) the parameter to take into account is the first one, while for tools that are intended to detect only penetrating attacks a good performance is achieved only with an high value for the second parameter along with a low value for the third parameter.

Since this work aims to evaluate tools that should be extensively used to detect SQL injection attacks and protect a web application, it would not be acceptable for the tools to produce a high level of false positives when browsed by non-malicious users. For this reason we first made some tests and preemptively tuned all the tools in order to make them not prone to false positive generation when exposed to a simple crawling of the web application. Although this influences the detection coverage, it is a realistic and fair starting point for the evaluation as all the tools are assessed based on similar assumptions.

Note that the analysis presented in this paper is not limited to the assessment of the metrics presented before. In fact, we also discuss in detail the strengths and limitations of each tool by exploiting the richness of the information made available by the Attack Injector for each of the attack performed. In practice, by looking at information about the types of attacks each tool is able to detect and the vulnerabilities and payloads injected, we could shape a more detailed profile of the capabilities of each tool.

IV. RESULTS AND DISCUSSION

In this section we present and analyze the results of the experiments. First, we present the outcomes obtained from the tools in each web application test. Then, we analyze in detail the detection patterns of each tool, underlining their strengths and weaknesses. Finally, we present some remarks about the lessons learned and a generic comparison of the different detection approaches.

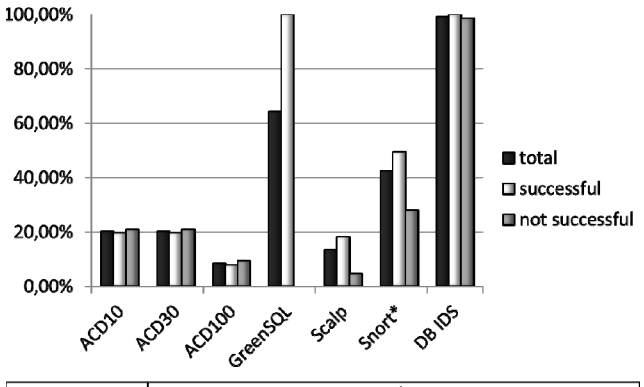
A. *Overall Results per Web Application*

1) *Tikiwiki*

For the TikiWiki web application, 120 attack attempts where performed over the 16 vulnerabilities injected by the Attack Injector. Of the 120 attempts, 77 where considered as being successful (i.e., penetrated the database) and 43 as not being successful. Fig. 4 summarizes the results for each tool (for ACD, three relevant values of the threshold where evaluated: 10, 30 and 100).

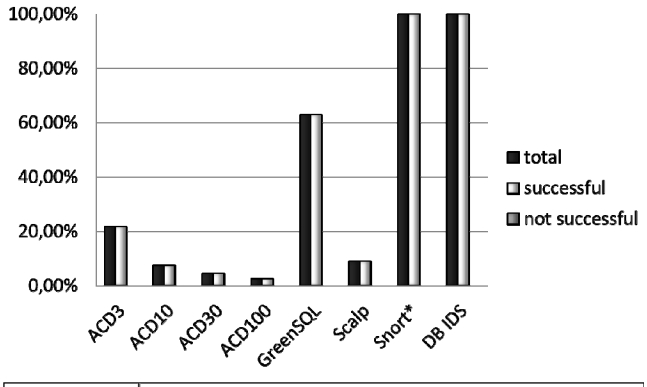
The most interesting result shown in Fig. 4 is that GreenSQL had, in this scenario, a perfect performance, as it detected all the successful attacks while it did not give alerts for the attacks that did not penetrate the database. Obviously, in this case the number of detections over the total attack attempts is not relevant.

The DB IDS produced very high coverage, close to 100% regarding the total set of attack attempts, but it failed to distinguish the successful attacks from the unsuccessful ones.



Attack Attempts	Tools						
	ACD 10	ACD 30	ACD 100	Green SQL	Scalp	Snort*	DB IDS
Total: 120	24/120	24/120	10/120	77/120	16/120	51/120	119/120
Successful: 77	15/77	15/77	6/77	77/77	14/77	38/77	77/77
Not Successful: 43	9/43	9/43	4/43	0/43	2/43	13/43	42/44

Figure 4. Detected attacks for Tikiwiki



Attack Attempts	Tools							
	ACD 3	ACD 10	ACD 30	ACD 100	Green SQL	Scalp	Snort*	DB IDS
Total: 245	53/245	18/245	11/245	6/245	154/245	22/245	245/245	245/245
Successful: 245	53/245	18/245	11/245	6/245	154/245	22/245	245/245	245/245
Not Successful: 0	0	0	0	0	0	0	0	0

Figure 5. Detected attacks for phpBB2

With the threshold set to 10 (the lowest one considered), ACD monitor only detected 20% of all attack attempts. Experiments conducted using lower thresholds (not shown in Fig. 4) produced very much higher detection rates but were not taken into account due to the very large amount of false positives reported, even when tested using non malicious traffic (i.e. crawling traffic). Given its detection approach, ACD is not able to distinguish between successful and not successful attacks. Moreover the attacks that are performed using POST HTTP requests remain undetected since in this type of attacks the malicious payload is not logged in the Apache access log. The higher values of the threshold that were evaluated produced even worst performance, achieving lower coverage values.

Scalp was able to detect only the attacks based on GET requests and that used two specific payloads: [or 'a'='a] and [67-ASCII('A')]. It totally ignored POST based attacks. In this way it managed to detect 13.33% of all the attack attempts.

Snort detected 42.5% of the attempts and the detections are biased towards the successful attacks (49.53% of the successful attacks were detected, while only 27.91% of the unsuccessful attacks triggered an alert). It must be noted that this detections are achieved only due to the use of the custom made rules for SQL injection. The standard rules provided with Snort produce no detection for these attacks.

2) phpBB2

For phpBB2 all the 245 attack attempts performed by the Attack Injector, exploiting 35 different injected vulnerabilities, were successful. For this reason, the results obtained with this application (Fig. 5) do not allow us to evaluate the ability of the tools to discriminate successful attacks from unsuccessful ones.

For ACD we considered four thresholds (3, 10, 30, 100). With the lowest threshold (3), the percentage of detected attack attempts was of 21.63%, but also in this case lower thresholds would have produced a higher level of coverage (but also very high level of false positives).

GreenSQL detected 62.86% of the attacks. Oddly it failed to detect all the attacks that were performed using the HTTP POST method. This fact is incidental and it is only due to the fact that these attacks have a pattern that is in the white list of GreenSQL for this application.

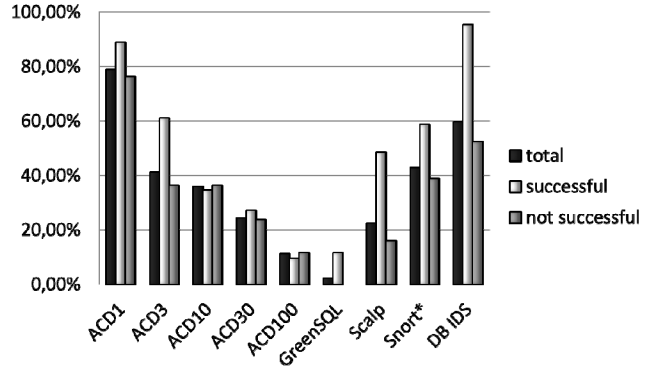
The DB IDS detected all the attacks for this application, but as aforementioned we cannot assess the ability of the tool to identify the successful ones.

Also in this case Scalp detected only the attacks that used [or 'a'='a] and [67-ASCII('A')] as payloads and that were associated with a GET request, failing to detect the same payload when embedded in a POST request.

Snort detected all the attacks but, again, all the alerts were from the custom made rules. Also in this case the default rules were not able to detect any of the SQL injection attacks.

3) MyReferences

In this application the Attack Injector was able to inject 33 vulnerabilities and attempted 686 attacks. Of these only 136 were considered successful. Fig. 6 summarizes the results for the different tools.



Attack Attempts	Tools								
	ACD 1	ACD 3	ACD 10	ACD 30	ACD 100	Green SQL	Scalp	Snort*	DB IDS
Total: 686	541/686	283/686	247/686	168/686	77/686	16/686	154/686	294/686	410/686
Successful: 136	121/136	83/136	47/136	37/136	13/136	16/136	66/136	80/136	129/136
Not Successful: 550	420/550	200/550	200/550	131/550	64/550	0/550	88/550	214/550	288/550

Figure 6. Detected attacks for MyReferences

GreenSQL had a very low coverage (11.76% of the successful attacks) but never raised an alarm for unsuccessful attacks. The low coverage was due to the fact that, for this application, GreenSQL was able to detect only two signatures: [or 1=1] and [or 'a'='a].

For this web application, the performance of DB IDS is better than the one achieved with TikiWiki. This time the DB IDS shows a clear bias in its detection towards the successful attacks. It detected 95.33% of the successful attacks, while only the 52.44% of the unsuccessful attacks generated alerts.

Snort detected only the attacks based on three payloads ([’], [or 'a'='a] and [67-ASCII('A')]), which corresponds to 42.86% of all the attack attempts. Anyway even if it cannot distinguish between successful and not successful attacks it generated alerts for 58.82% of the successful attacks and only 38.9% for the unsuccessful ones.

Since the character distribution of the HTTP requests for this application is less variable, the acceptable ACD thresholds are lower, which enhances the performance of the tool. Again, attacks made trough POST are invisible to ACD monitor. With the threshold set to 1 (the lowest one), the overall coverage (for both successful and unsuccessful attacks) reached 78.96%. Anyway, ACD monitor cannot separate HTTP requests that penetrate the web application from those that do not and this is proved by the fact that the detection percentage is almost the same for both successful and not successful attacks.

Scalp detection pattern is the same as for the other two applications. It can only detect GET attacks associated with the [or 'a'='a] or [67-ASCII('A')] payloads. The achieved detection coverage is in this case 22.45%.

Snort, using the custom rules, detected all the attacks containing three of the payloads ([’] [or 'a'='a] [67-ASCII('A')]), but none containing the other payloads. Similarly to the TikiWiki scenario, detections were higher for successful attacks (58.82%) than for the unsuccessful ones (38.91%).

B. Detailed Tools Analysis

1) GreenSQL

The first aspect to be mentioned about GreenSQL is that it is the only tool that has not generated any alerts for not successful attack attempts. This makes it the more accurate tool among those we tested in the sense that it is the only one that can accurately discriminate between successful and not successful attacks. However it is far from being perfect. In fact, as shown in Table III, the successful attacks coverage varies significantly between the different web applications we tested.

These very different behaviors are related to the detection approach that GreenSQL uses: during the learning phase GreenSQL identifies patterns of queries associated to the normal operation of the web application, populating a white list of queries considered non malicious. When a query is executed on the database protected by GreenSQL, it is evaluated against the patterns in that white list and if it does matches one of the existing patterns then it is considered non malicious. For this reason, only the queries that do not match these patterns are taken into account. In MyReferences only some of the payloads containing an “OR” succeed in modifying the query structure and so escaped from the white list and are detected. In phpBB2 and TikiWiki the situation is more complex, since the number of queries and the structure complexity increase significantly. In phpBB2 all detections are for queries that use sensitive tables, while for TikiWiki there are detections also for the use of “OR” tokens.

TABLE III. GREENSQL COVERAGE RESULTS

Web Application	Successful Attacks Coverage
TikiWiki	100.00%
phpBB2	62.86%
MyReferences	11.76%

2) DB IDS

The database IDS is an anomaly based IDS, thus we were not expecting this tool to be very accurate. In fact, although it showed a very high coverage in terms of the total number of attacks, the number of unsuccessful attacks that were reported is also very high. Anyway we must point out that for the MyReferences application it was able to better recognize successful attacks rather than unsuccessful attempts. This is probably due to the fact that this application has a very simple architecture, which facilitates the task of identifying the deviation from the normal behavior.

3) ACD monitor

To better understand the performance of the Anomalous Character Distribution (ACD) monitor we must underline that it is an application level tool that only uses the access log as source for detection. For this reason, this tool is intrinsically not able to determine if an attack is successful or not. This means that, to evaluate its performance, we must look only to the total coverage that it achieves, not taking into account the successful/unsuccessful attack coverage. In fact, it is clear in the results presented before that the percentage of attack attempts that generated detection is always almost the same in both the successful and not successful attacks. Table IV reports the coverage achieved by the tool taking into account all the attack attempts generated by the Attack Injector.

Another important issue related to the use of the access log as input is related to the fact that attacks where the malicious payload is injected trough a POST produce a request in the log that is undistinguishable from a legitimate one. This way, this type of attacks is invisible to this tool.

Another key characteristic of this tool is that its performance is strongly influenced by the learning phase and by the threshold that is used to discriminate between malicious and non-malicious requests. For this reason, the tool is much more efficient in the MyReferences web application than in the other two applications. That is due to the fact that MyReferences has a much more simple structure

TABLE IV. ACD MONITOR BEST COVERAGE THRESHOLD RESULTS

Web Application	All attack attempts Coverage	Threshold used	Number of accessible PHP files
Tikiwiki	20.00%	10	23
phpBB2	21.63%	3	12
MyReferences	41.25%	1	11

and is composed by a limited number of PHP files. This leads the HTTP requests for this application to be much less variable, allowing lower thresholds and making simpler the identification of the anomalous requests.

4) Apache Scalp

Apache Scalp is the tool that reported the worst results during our experiments. Table V summarizes its coverage.

The detection pattern for Scalp is very simple: it just detected all the attacks based on two specific payloads ([or 'a'='a] and [67-ASCII('A')]) that were injected using GET requests. As for ACD monitor the POST attacks are invisible to this tool. We must also notice that Scalp uses signatures of SQL Injection attacks that are usually targeted for malicious payloads much more elaborated than then those applied by the Attack Injector. As an example consider the following two HTTP requests that trigger a SQL Injection attack detection by Scalp:

- `GET /index.php?option=com_portfol&Itemid=1&task=viewcategory&vcatid=-1+union+select+concat(username,char(58),password)KHG+from+jos_users—`
- `GET /index.php?option=com_equotes&id=13%20and%201=1%20union%20select%20user(),concat(username,0x3a,password),user(),user(),user(),user(),user()%20FROM%20jos_users—`

and compare them with two examples of not detected attacks performed by the attack injector:

- `GET /html/tikiwiki-1.9.9/tiki-editpage.php?page=sandbox%20'`
- `GET http://192.168.189.146/html/phd-mysql-test/edit_authors.php?id=208%20'&name=&submit_authors=Insert`

As we can see, Scalp is not capable of detecting the simple attempts a hacker usually makes to explore the hidden mechanisms of a web application, nor the preliminary attacks that precede a more dangerous penetrating attack. Anyway it can detect elaborate attacks that would lead a deep penetration in the database. Results presented in [25] show that Scalp achieved much better results when exposed to elaborated attacks.

5) Snort

Snort also achieved very weak results. Basically the standard rules provided with snort produced no detection when analyzing both HTTP and SQL traffic. This is particularly negative as Snort was the only tool capable of analyzing what was happening on both the web and database tiers of the application.

Again, the signature-based approach should be held responsible for the weak performance of the tool. In fact, most of the signatures available in Snort are signatures of very specific and well-known vulnerabilities for specific web applications. For example, the following signature is intended to detect a specific vulnerability of the tag_board.php file present in some versions of phpBB:

```
alert tcp $EXTERNAL_NET any -> $HOME_NET
$HTTP_PORTS (msg:"WEB-PHP phpBB mod tag board
sql injection attempt";
flow:established,to_server;
uricontent:"tag_board.php"; nocase;
uricontent:"action=delete"; nocase;
uricontent:"id="; nocase;
pcre:"/tag_board.php\x3F[^\r\n]*action=delete
[^\r\n]*id=[^\r\n\x26]*(select|insert|delete)
/Usmi"; threshold:type limit, track by_src,
count 1, seconds 300; metadata:policy
security-ips alert, service http;
reference:bugtraq,32701; reference:cve,2008-
6314; classtype:web-application-attack;
sid:15425; rev:1;)
```

Obviously this approach cannot detect attacks like those performed by the Attack Injector that artificially creates, injects and attacks new vulnerabilities on the fly. On the other hand, by using those custom made rules the detection approach reverts to something very similar to an application level tool, since these rules basically look for some character patterns, usually associated with SQL injections, in the HTTP traffic. Indeed with this type of rules snort operates in a way very similar to Scalp. The much higher coverage of Snort with respect to Scalp is only due to the fact that Snort has a signature for one more payload (['']), and because POST attacks are not invisible to the network level sniffer.

TABLE V. SCALP COVERAGE RESULTS

Web Application	All attack attempts Coverage
Tikiwiki	13.33%
phpBB2	8.97%
MyReferences	22.45%

C. Lessons Learned

In this section we summarize the results of the analysis made in the previous sections and provide a few general remarks.

1) Application vs Database Level

One aspect that is evident from our experiments is that by looking to the **database level** it is possible to better understand what is going on in the web application. Without a tool monitoring this level it is impossible to detect if a piece of malicious SQL has penetrated the application and reached the database. For the same reason we must acknowledge that **application level** tools can be fooled really easily in detecting nonexistent attacks (both Scalp and ACD monitor are prone to this). A smart attacker could make one of these tools to generate an unlimited number of alerts just using a well-furnished repository of well-known attack signatures.

2) Network level

Snort is the only network level tool we tested. We must report that it failed to exploit the potentiality of the privileged point of view the network level offers, allowing hypothetically monitoring both the HTTP and MySQL traffic. Anyway we think that rules like the ones we tested in this work [23], based on more generic patterns and not

limited to the identification of signatures of specific vulnerabilities should be created to increase HTTP level detections and to provide some improved detection capabilities for the database level for SQL Injection attacks.

3) Signature Based vs Anomaly based

Results suggest that, in general, **anomaly-based** tools (like ACD monitor and DB IDS) present better performance when working with simpler applications (like MyReferences), since the characterization of the normal behavior is easier and the detection of variation from this profile is more accurate. However, for more complex applications (like TikiWiki), the **signature-based** tools (like GreenSQL, Scalp and Snort) are the ones that perform better, while anomaly based detection either has very low coverage (like the ACD monitor) or just fails to distinguish successful attacks from the unsuccessful ones. This is the case of the DB IDS when dealing with TikiWiki, in which the tool detects all the successful attacks but generates alert also for almost all non-penetrating attacks.

V. Conclusions and Future Work

In this paper we presented the results of an experimental study on the effectiveness of five security tools for the detection of SQL Injection attacks. These tools comprise both well-known system administrator instruments and innovative detection tools proposed by researchers. They can be used to monitor web applications at different architectural levels (Application, Database and Network) and use different approaches for the detection procedure.

The evaluation experiments were performed using three web applications, each one characterized with a different size and complexity, which were exploited using an Attack Injection technique. This technique allowed us to artificially inject realistic vulnerabilities into the code of the web applications and then automatically attack these vulnerabilities, attempting to inject malicious payloads in the database.

Results of the experiments highlighted the low performance of the tools as no tool did perform well in all the different situations. Additionally, we have shown how the performance achieved depends on many factors, but in particular on the architectural level that is monitored by the tool, on the detection approach used, on the types of attacks performed, and on the complexity of the target web application.

For this reason we believe that Attack Injection can be a very useful instrument to assess the detection ability of intrusion detection tools in specific contexts and for specific web applications. In practice, the technique can be extremely useful to assess the level of trust a system administrator can have about the security tools installed to protect his system.

Future work includes creating the instruments needed to fully automate all the steps of the evaluation procedure that was used in our experiments. The aim should be to create a standard benchmark procedure for using Attack Injection as an advanced assessing instrument of the detection performance of security tools deployed in a web serving system.

References

[1] Acunetix Ltd, Feb. 2007, http://www.acunetix.com/news/security-audit-results.htm

[2] S. Christey, "Unforgivable Vulnerabilities", The MITRE Corporation, Black Hat Briefings, August 2007

[3] J. Williams, D. Wichers, "OWASP top 10 – 2010", OWASP Foundation, Apr. 2010

[4] D. Stuttard and M. Pinto, "The Web Application Hacker's Handbook: Discovering and Exploiting Security Flaws", Wiley Publishing Inc., 2007

[5] Symantec. "Symantec Report on the Underground Economy", 2008.

[6] P. Herzog, "OSSTMM 2.2. Open-Source Security Testing Methodology Manual", 2nd ed., ISECOM, 2006

[7] M. Howard and D. LeBlanc, "Writing secure code", Microsoft Press, 2003

[8] R. Richardson, "2008 CSI Computer Crime & Security Survey", Computer Security Institute, 2008.

[9] R. Richardson and S. Peters, "2009 CSI Computer Crime & Security Survey", Computer Security Institute, 2009

[10] J. Calderón, "Preparing a Strategy for Application Vulnerability Detection: Setting the basis to secure critical information assets", Softtek, 2009.

[11] J. Arlat, A. Costes, Y. Crouzet, J.-C. Laprie and D. Powell, "Fault injection and dependability evaluation of fault-tolerant systems", IEEE Trans. on Computers, Aug. 1993

[12] J. Fonseca, M. Vieira and H. Madeira, "Vulnerability & attack injection for web applications", International Conference on Dependable Systems and Networks, Estoril, Lisbon, Portugal, 2009.

[13] D. Dougherty, 2001, http://onlamp.com/pub/a/onlamp/2001/01/25/lamp.html

[14] Nexen.net, Feb. 2008, http://www.nexen.net/chiffres_cles/phpversion/18360-php_stats_evolution_for_april_2008.php

[15] TikiWiki, May 2010, http://tikiwiki.org/

[16] phpBB, May 2010, http://www.phpbb.com/

[17] E. Biermann, E.Cloete, and L. Venter, "A comparison of intrusion detection systems", Computers and Security, 2001, pp. 676–683.

[18] apache-scalp, Apache log analyzer for security, May 2010, http://code.google.com/p/apache-scalp/

[19] C. Kruegel and G. Vigna, "Anomaly detection of web based attacks", Proc. of the 10th ACM conference on Computer and Communication Security (CCS'03), ACM Press, Oct. 2003, pp. 251-261.

[20] "GreenSQL, an Open Source database firewall used to protect databases from SQL injection attacks", May 2010, http://www.greensql.net

[21] M. Roesch, "Snort - Lightweight Intrusion Detection for Networks", Proc. of the USENIX LISA '99 Conference, Nov. 1999.

[22] Snort, an open source network intrusion prevention and detection system (IDS/IPS), May 2010, http://www.snort.org/

[23] K. Mookhey and N. Burghate, "Detection of SQL Injection and Cross-site Scripting Attacks", Mar. 2004, http://www.securityfocus.com/infocus/1768

[24] J. Fonseca, M. Vieira and H. Madeira, "Detecting Malicious SQL", Int. Conference on Trust, Privacy & Security in Digital Business, Sep. 2007.

[25] L. Coppolino, S. D'Antonio, I. A. Elia, L. Romano, "From Intrusion Detection to Intrusion Detection and Diagnosis: An Ontology-Based Approach", Software Technologies for Embedded and Ubiquitous Systems, Volume 5860/20090020– ISSN 0302-9743, 2009, pp. 192-202