Assessing the Impact of Firewalls and Database Proxies on SQL Injection Testing

Dennis Appelt(✉), Nadia Alshahwan, and Lionel Briand

Interdisciplinary Centre for Security, Reliability and Trust,
University of Luxembourg, Luxembourg, Luxembourg
{dennis.appelt,nadia.alshahwan,lionel.briand}@uni.lu

Abstract. This paper examines the effects and potential benefits of utilising Web Application Firewalls (WAFs) and database proxies in SQL injection testing of web applications and services. We propose testing the WAF itself to refine and evaluate its security rules and prioritise fixing vulnerabilities that are not protected by the WAF. We also propose using database proxies as oracles for black-box security testing instead of relying only on the output of the application under test. The paper also presents a case study of our proposed approaches on two sets of web services. The results indicate that testing through WAFs can be used to prioritise vulnerabilities and that an oracle that uses a database proxy finds more vulnerabilities with fewer tries than an oracle that relies only on the output of the application.

Keywords: SQL injections · Blackbox testing · Web services

1 Introduction

In recent years, the world wide web evolved from a static source of information to an important application platform. Banking, shopping, education, social networking and even government processes became available through the web. The rise of cloud-backed applications and web-centric operating systems like Windows 8 or Chrome OS further accelerated this shift.

The popularity of web applications can be attributed to their continuous availability, accessibility and flexibility. However, this also caused the web to become a target for malicious attackers. Recent studies found that the number of reported web vulnerabilities is growing sharply [11]. Web applications experience, on average, 27 attacks per hour [4].

Web technologies, such as HTML5, are constantly developing to enable the production of richer web applications and enhance user experience. However, with new functionality comes a higher risk of introducing vulnerabilities [15,22]. Developers might be unaware of the newest security concepts and unintentionally introduce risks to their applications. Attackers, on the other hand, might misuse new features of web technologies to compromise previously secure applications. These risks raise the need for systematic and well-defined security testing approaches that can cope with this constant evolution in security risks. Moreover, with the high pressure of deadlines and limited time and resources allocated to testing, these approaches need to be automated as well as accurate and effective at detecting vulnerabilities.

In addition to testing, practitioners might use several run-time protection mechanisms to protect their applications against attacks, such as Web Application Firewalls (WAFs) and database proxies. WAFs monitor input values received by the application for attack strings while database proxies monitor the communication between the application and the database for suspicious SQL statements. We believe that these two technologies can be utilised in the security testing process and can also affect the results of evaluating different techniques.

In this paper we assess the impact of using WAFs and database proxies on testing for SQL injection vulnerabilities, which are one of the most widely spread types of vulnerabilities [5,24]. We propose that WAFs can be used to prioritise vulnerabilities by focusing developers effort on vulnerabilities that are not protected by the WAF first. We also investigate the effectiveness and efficiency of using database proxies as oracles for SQL injection testing instead of just relying on the output of the application. We expect that using database proxies would enhance the detection rates of vulnerabilities.

The results of our case study on two service-oriented web applications with a total of 33 operations and 108 input parameters indicates that using database proxies as an oracle does indeed improve detection rates. The results also show that detecting vulnerabilities while testing through a WAF is more challenging and that many vulnerabilities are protected by the WAF. This indicates that testing through the WAF can be used to prioritise fixing vulnerabilities in practice.

The rest of this paper is organised as follows: Sect. 2 provides a background on SQL injection testing and presents the definitions of the terms that are used in this paper. Section 3 discusses WAFs and database proxies, whilst Sect. 4 reviews related work. Section 5 presents the case study together with a discussion of results and threats to validity. Finally, Sect. 6 concludes and explores future work.

2 SQL Injection Testing

Existing injection testing approaches can be classified into two main categories: White-box and black-box approaches. These approaches try to detect vulnerabilities caused by poorly validated inputs that an attacker might exploit to cause the application to behave unexpectedly or expose sensitive data. For example, the attacker can construct harmful strings that flow into SQL statements and change their intended behaviour.

White-box approaches use static and dynamic analysis of the source code to detect vulnerabilities. Some White-box approaches track the execution of the program to identify un-validated inputs that flow into SQL statements or output commands [19]. Other approaches use symbolic execution to identify the constraints that need to be satisfied to lead to an SQL injection attack [12]. Shar and Tan [23] used data mining of the source code to predict vulnerabilities.

White-Box approaches require access to the source code of the application, which might not always be possible. Many companies outsource development of their systems or acquire third party components. Although these companies do not have access to the source code, they still need to ensure that their software is secure.

Black-box techniques typically explore a web application to find input fields that are then used to submit malicious inputs to the application. The output is analysed to determine if the attack was successful. Malicious inputs are formed, for example, using fuzzing techniques that generate new inputs from existing patterns of known vulnerabilities [16]. The detection rates of black-box techniques depend significantly on the ability to craft effective inputs.

SQL injection vulnerabilities are one of the most critical and widely spread types of vulnerabilities [5,24]. An attacker targeting this type of vulnerabilities attempts to manipulate the values of input parameters of an application to inject fragments of SQL commands that evade security mechanisms and flow into an SQL statement. The goal is to alter the SQL statement and change its behaviour in a way that could benefit the attacker. For example, the additional SQL code might result in an SQL query returning more data than what was intended by the developer. Such attacks are usually possible because some developers insert the values provided by the user into SQL statements using string concatenation. If these input values are not validated and checked properly for SQL attack patterns, the SQL code they contain will be part of the SQL statement making it possible to change the effect of the statement.

2.1 Definitions

In this section, we precisely define the terms related to SQL injection testing that will be used throughout this paper. Understanding the meaning of terms such as vulnerability, detectable and exploitable might seem intuitive. However, their precise definition might influence the interpretation of results and evaluation of testing techniques. To the best of our knowledge, these terms have not been defined formally in previous research on SQL injection testing.

In most cases, when input values are used in SQL statements, their values are used as data and not as part of the SQL code to be executed. For example, the following SQL statement is a simplified version of a statement found in one of the applications we use in our case study:

```
$sql="Select * From hotelList where country ='".$country."'";
```

The value of the input parameter `$country` is used to limit the rows returned by the SQL query to hotels located in the country provided by the user. If this input is not properly validated and checked for malicious values, a user can provide an input such as:

```
    ' ; drop table hotelList;--
```

The result of concatenating this input with the previous SQL statement would be:

```
$sql = "Select * From hotelList where country =" ;
              drop table hotelList;--';
```

When the database server executes this SQL code, the `Select` statement would return no values while the `drop table` statement would delete the table hotelList (if permission to drop tables is not configured correctly on the database level to prevent such actions). The rest of the command will not be executed because it is commented (`--` symbol). The input in this case was interpreted as SQL code rather than data, allowing the user to alter the database causing loss of information and unavailability of the system because the table was deleted. We can define an SQL vulnerability as follows:

Definition 1. *An SQL vulnerability in a system under test is an input parameter where part or all of its value is used in an SQL statement and interpreted as SQL code instead of treated as data in at least one execution of the system.*

Whether all or part of the input parameter value is interpreted as SQL depends on the attack string used and the logic of the application. For example, in the previous SQL statement if the attack string was `Luxembourg'; drop table hotelList;--` then part of the input value (`Luxembourg`) is treated as data while the rest (`drop table hotelList;--`) is interpreted as SQL.

The goal of an SQL injection testing approach is to detect vulnerabilities. Whether a vulnerability is detected or not also depends on the oracle used. Therefore, we can define a detectable vulnerability as follows:

Definition 2. *A detectable SQL vulnerability with regards to an oracle is a vulnerability that can be detected by this oracle.*

In some cases, a vulnerability exists in the system but an attacker might not be able to exploit it. For example, a numeric input might not be validated properly to ensure that only numeric data can be assigned to it. However, a firewall might be configured correctly to block any attack strings submitted to that same input. Therefore, although the attacker can submit string values to this numeric input parameter, which is unintended behaviour, the attacker would not be able to use this vulnerability to gain any benefit. We can define an exploitable SQL vulnerability as follows:

Definition 3. *An exploitable SQL vulnerability is a vulnerability that can be used to cause an information leak, an unauthorised change in the state of the database or system or causes the system to be unavailable.*

Information leakage, changes to the state or system unavailability might not be the only negative effects an attacker can inflict on the system. However, this definition can be extended when needed to include other types of harmful effects. In some cases, deciding if a vulnerability is exploitable can not be done

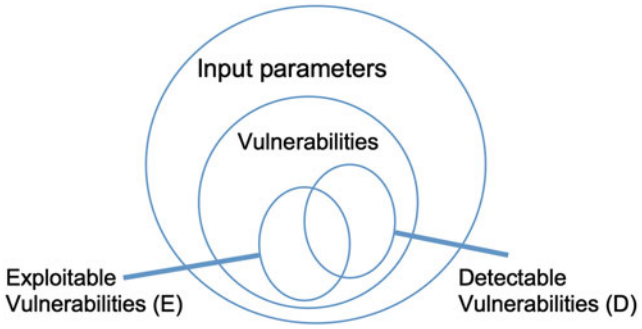
Input parameters
Vulnerabilities
Exploitable Vulnerabilities (E)
Detectable Vulnerabilities (D)

Fig. 1. The different classifications of input parameters to illustrate the relationship between the different classifications.

automatically and requires manual inspection by engineers who have the domain knowledge to decide if a vulnerability is exploitable. For example, a vulnerability in the system might be exploited to leak information but the information that is leaked is not sensitive or can be obtained by any user through alternative methods. However, an automated approach might be able to estimate the probability of a vulnerability being exploitable based on some heuristics. Vulnerabilities can then be ranked based on this probability to reduce the time required for manual inspection or focus efforts on vulnerabilities that might pose a higher threat.

Figure 1 illustrates the relationship between the different classifications of input parameters. A subset of all input parameters might be vulnerable, while a subset of those vulnerabilities is exploitable. Detectable vulnerabilities could either be exploitable or not exploitable. The intersection of detectable vulnerabilities and exploitable vulnerabilities ($E \cap D$) is the set of critical security faults that the testing process found. The set of exploitable vulnerabilities that are not detectable ($E-D$) represents the false negatives of the testing process.

3 Security Mechanisms

Several types of security mechanisms are used at run-time to protect against SQL injection attacks. The two main security mechanisms used in practice are firewalls and database proxies. In the next two sections we examine each type in more detail and discuss their effect and potential utilisation in SQL security testing.

3.1 Web Application Firewalls

A Web Application Firewall (WAF) examines every request submitted by the user to the application to decide if the request should be accepted (when it is a legal request) or rejected (if the request is malicious). The WAF makes this decision by examining each input value in the request and checking if the value matches an attack pattern typically using a set of rules (regular expressions).

The performance of the WAF and the protection it provides depends on this set of rules. Since these rules are created and maintained manually by the application owner, they might be error-prone and security holes might be introduced by mistake. On the other hand, attackers are continually searching for ways to evade firewalls by using mechanisms such as obfuscation where equivalent encodings of attack patterns are used that might not be recognized by the WAF. WAFs are commonly used in the industry, for example, using a WAF is necessary to be compliant with the Payment Card Industry Data Security Standard [20] for systems that use/process credit cards.

Naturally, using a WAF affects the security assessment of an application; some input parameters might be vulnerable if the application is accessed directly but not vulnerable when a WAF is used. For example, an input parameter that flows to an SQL statement might not be validated for SQL injection attack patterns but the WAF is configured to detect and reject such attack patterns. In some cases in practice, all validation and filtering might be delegated to the WAF. Testing applications with such a set-up using approaches that only take into account the application itself and not the WAF might result in determining that all inputs that are used in SQL statements are vulnerable. Ideally all these vulnerabilities should be addressed to provide two layers of protection in case an attacker is able to bypass the WAF. However, with limited time and resources dedicated to testing, which is often the case in the industry, test engineers might want to focus on fixing vulnerabilities that can still be exploited even when using a WAF. Therefore, testing an application for SQL injection vulnerabilities through a WAF can be used to identify and prioritise vulnerabilities that can be detected through the WAF.

Testing through a WAF can also have other useful applications, such as testing the WAF itself. This can be useful, for example, if a choice needs to be made by the application owner between different alternative WAFs. The application can be tested using each WAF and the firewall that provides the most protection can be chosen. Finally, testing the WAF could help in evaluating and refining its rule set. When a vulnerability is found that can be detected while using a WAF, the developers, after fixing the application code to eliminate the vulnerability, can define new rules or adjust existing rules to protect the application against similar types of vulnerabilities. This might be useful to protect the application against similar types of vulnerabilities that might be introduced in subsequent versions of the system.

3.2 Database Proxies

Database proxies (e.g., GreenSQL [13], Snort [21], Apache-scalp [2]) reside between the application and the database and monitor each SQL statement issued from the application to the database for malicious commands. These proxies have an advantage over WAFs in that they have access to the SQL statement after it is formulated and, therefore, have more information to decide if an SQL command is an attack. Database proxies can usually be configured to either a prevention mode where malicious attacks are blocked by the proxy or a monitoring mode where suspicious requests are allowed to execute but logged for further examination by an administrator.

Typically, database proxies use either a risk-based or learning-based approach to decide if an SQL statement is malicious. The risk-based approach assigns a risk score to each intercepted SQL statement which reflects the probability of the statement being malicious. To calculate the risk score, each statement is assessed for SQL fragments frequently used in SQL injection attacks, e.g. the comment sign or a tautology. In the learning based approach, the security engineer first sets the proxy to a learning mode and issues a number of legal requests that represent the application's behaviour. The proxy, thereby, learns the different forms of SQL commands that the application can execute. When the proxy is set to the monitoring or prevention mode, any request that does not comply with these learnt forms is flagged as malicious. The effectiveness of the proxy is dependent on this learning phase: If the requests issued in this phase do not represent all legal behaviour of the application, legal requests might be flagged as suspicious when the proxy is used in practice (high false positive rate).

Existing black-box SQL injection testing approaches commonly use an oracle that relies on the output of the application to decide if a vulnerability was detected [1,6,17]. In this paper we propose using a database proxy as an oracle. Since proxies have access to the SQL statement after all input values have been inserted in the statement and all processing is done, we expect that using the proxy as an oracle would enhance detection rates.

4 Related Work

In this section we briefly review existing techniques for black-box SQL injection testing and also review the results of empirical studies that compare different black-box testing techniques.

Huang et al. [17] proposed a black-box SQL injection approach that learns the application's behaviour and then compares this to the behaviour of the application when SQL injection attacks are submitted. Antunes and Vieira [1] use a similar oracle but focus on SQL and server errors rather than the whole output. For example, if the legal test case led to an SQL or server error but the attack was successful, the approach infers that a vulnerability was found since the attack was able to circumvent the checks that caused the original error. Ciampa et al. [6] analyse the output, including error messages, of both legal and malicious test cases to learn more about the type and structure of the back-end database. This information is then used to craft attack inputs that are more likely to be successful at revealing vulnerabilities. These approaches use an oracle that relies on observing and analysing the output, while we propose using a database proxy as an oracle to enhance detection rates.

Several empirical studies evaluated and compared commercial, open-source and research black-box SQL injection testing tools [3,9,25]. These studies found that black-box testing tools have low detection rates and high false positive rates for SQL injections. This result highlights the need to improve both test generation approaches and oracles for SQL injection testing. In this paper, we focus on improving the oracle by using database proxies rather than relying on the output of the application.

Elia et al. [10] evaluated several intrusion detection tools, including the database proxy GreenSQL that we use in this paper. The study injects security faults into the applications under study and then automatically attacks the application to evaluate the effectiveness of the intrusion detection tools studied. These papers focused on testing and comparing security mechanisms, such as WAFs and database proxies, while we propose utilising these tools in the security testing process.

5 Case Study

We designed the case study to answer the following research questions:

RQ1: What is the impact of using an oracle that observes communications to the database on SQL injection vulnerability detection?

We expect that an oracle that observes the database to determine that a vulnerability was detected might improve the detection rates of an SQL injection testing approach compared to an oracle that only relies on the output. However, using such an oracle might result in a high number of false positives or have other implications on the results. To answer this question, we conduct an experiment where we perform SQL injection testing using a state-of-the-art tool that relies only on the output and a prototype tool that we developed that uses a state-of-the-art database proxy as an oracle. We compare the number of vulnerabilities detected and the number of test cases that needed to be generated before the vulnerability was detected. We also examine the requests that detected vulnerabilities for both approaches to investigate whether they led to the formulation of executable malicious SQL statements and, therefore, led to detecting exploitable vulnerabilities.

RQ2: How does testing the web services directly and testing them through a WAF impact the effectiveness of SQL injection testing?

Generating test cases that are able to detect vulnerabilities in web services through a WAF is naturally expected to be more challenging than testing the application directly, since the WAF provides an additional layer of protection. However, vulnerabilities that can be detected while testing through the WAF pose a more pressing threat since they are completely not protected. To answer this question, we test the application using the two testing approaches (the state-of-the-art tool and our prototype tool) through a state-of-the-art WAF and compare the results to those obtained without using the WAF. A reduction in the number of vulnerabilities found might indicate that testing through the WAF can be used to prioritise fixing vulnerabilities that are not protected by the WAF. Such reduction might also indicate that we need more advanced test generation techniques for security testing that can penetrate the more sophisticated protection techniques of WAFs and identify harder to detect vulnerabilities.

Table 1. Details about the two applications we used in the case study

Application	#Operations	#Parameters	LoC
Hotel reservation service	7	21	1,566
SugarCRM	26	87	352,026
Total	33	108	353,592

5.1 Case Study Subjects

We selected service-based web applications rather than traditional web applications to eliminate the effects of crawling the web application on results. Web services have well-defined and documented APIs that can be used to call the different operations in the application. On the other hand, web applications require a crawling mechanism to be built into the testing technique to explore the application and find input fields that might be vulnerable. The crawling mechanism might impact the effectiveness of the overall testing approach as noted by previous studies [3,9,18].

We chose two open-source service-based applications as subjects for the case study. Table 1 provides information about the number of operations, input parameters and lines of code for the chosen applications. The Hotel Reservation Service was created by researchers[1] to study service-oriented architectures and was used in previous studies [7]. SugarCRM, is a popular customer relationship management system (189+K downloads in 2013[2]). Both applications are implemented using PHP, use a MySQL database and provide a SOAP-based Web Service API.

5.2 Prototype Tool

We developed a prototype tool in Java that uses a set of standard attacks as test cases and a state-of-the-art database proxy as an oracle to help answer our research questions. Specifically, the tool is expected to help verify that an oracle which observes database communications to detect SQL injection vulnerabilities could improve the detection rate of an SQL testing approach. The architecture of the tool is depicted in Fig. 2. The testing process can be divided into three subprocesses: test case generation, delivery mechanism and vulnerability detection.

The *test case generation* process takes a valid test case as input (a test case where input data conforms to the specification of the operation under test). This valid test case is transformed into a malicious test case by replacing one input parameter value at a time with an SQL injection attack chosen from a list of standard attacks. We provided the tool with a list of 137 standard attacks that was compiled by Antunes and Vieira [1] and represents common SQL injection attacks. A parameter is replaced with an SQL attack from the list until a vulnerability is detected or all attacks are used. This process is repeated for each input

[1] http://uwf.edu/nwilde/soaResources/
[2] http://sourceforge.net

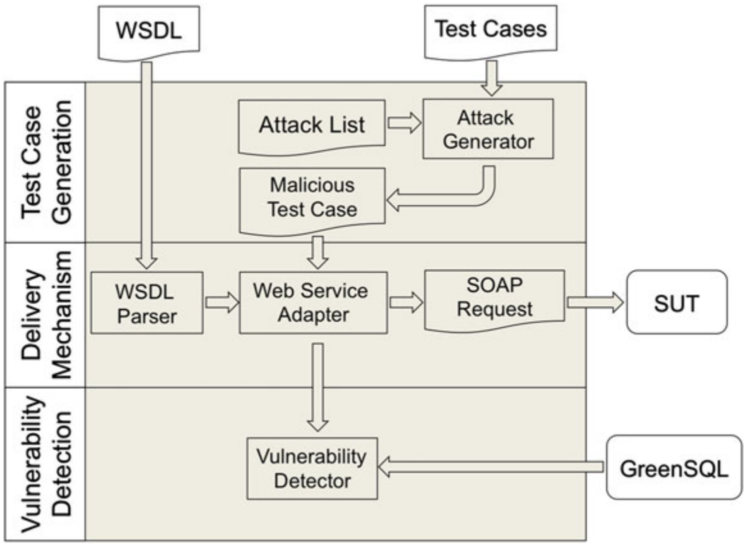

Fig. 2. Architecture of the prototype testing tool.

parameter of the operation under test. The *delivery mechanism* process encapsulates all implementation details which are necessary to deliver the malicious test case to the SOAP-based Web Service and obtain a response. The *vulnerability detection* process uses the state-of-the-art database proxy *GreenSQL* as an oracle to detect vulnerabilities.

GreenSQL is an SQL injection attack detection and prevention tool that supports both the learning-based approach and the risk-based approach discussed in Sect. 3.2. In our case study, we used the learning-based approach to detect malicious SQL statements. We chose GreenSQL based on the results of a previous study that compared GreenSQL to five similar tools and found it to be the most effective in detecting SQL injection attacks [10].

5.3 Case Study Set-up

To perform the case study, we conducted two sets of experiments. In the first set of experiments (Fig. 3), we applied our prototype tool and a state-of-the-art black-box security testing tool that relies only on the output of the application to the two case study subjects. We selected *SqlMap* [8] as a representative for traditional black-box testing tools that rely only on the output. We chose SqlMap because it is an open source free tool that provides support for testing web services as well as web applications. SqlMap is also one of four tools listed on the Open Web Application Security Project (OWASP) website [24] for automated SQL injection testing. The tool was also used in previous studies [6,14].

In the second set of experiments (Fig. 4), we applied the same two tools to our case study subjects but tested the applications through a WAF to answer RQ2. We selected *ModSecurity* for our experiments. ModSecurity is a WAF that

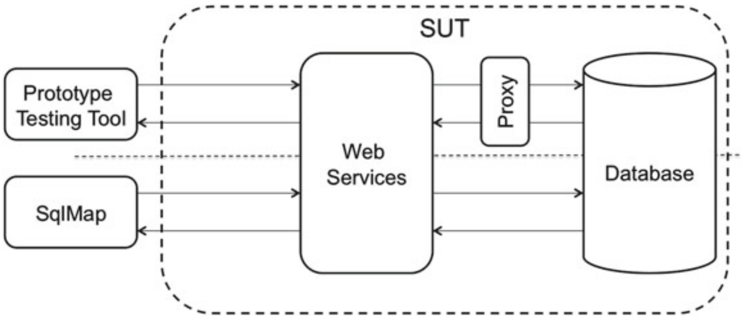

Fig. 3. Experimental set-up for RQ1: The effect of observing database communications on detection rates.

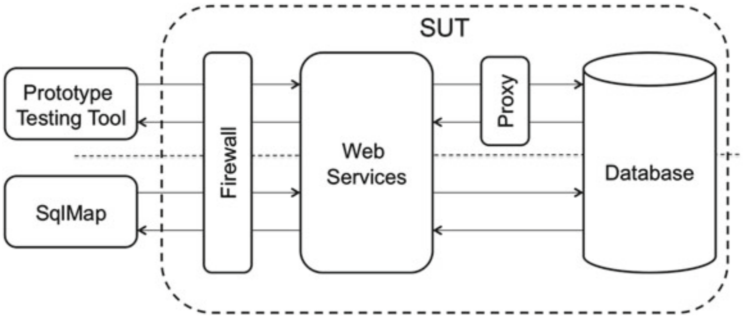

Fig. 4. Experimental set-up for RQ2: The influence of testing through a Web Application Firewall.

protects web servers (Apache, IIS, Nginx) from common threats including SQL injections. Since ModSecurity requires a rule set to identify and reject attacks as discussed in Sect. 3.1, in our case study we used the OWASP [24] core rule set (version 2.2.7).

Our prototype tool is deterministic, therefore we ran the tool once on each application and for each set-up. SqlMap, on the other hand, is not deterministic, therefore we ran the tool 30 times for each application and each set-up. We used the default configuration for SqlMap, except for a minor modification that omits test cases that cause the database to pause the execution of a query, for example by calling the `sleep()` operation, to avoid long execution times.

5.4 Results

This section discusses the results of running the experiments that we described in the previous section on each of the two case study applications. For each experiment, we counted the number of vulnerabilities found by each tool and the number of tries (requests) that the tool needed to generate and execute before finding each vulnerability. As we mentioned before, SqlMap is non-deterministic,

Table 2. Collected data for the described experiments.

Application	Firewall	Prototype		SqlMap	
		#Vulner.	avg. # tries	#Vulner.	avg. # tries
Hotel reservation Service	Without	6	6.3	6	1,306.55
	With	6	28	0	–
SugarCRM	Without	6	2	3.87	566.77
	With	3	34	0	–

therefore, we repeated any experiments that involve it 30 times and calculated the average number of vulnerabilities and tries.

Table 2 summarises all the results obtained from our experiments. Both the prototype tool and SqlMap were run on the two web service applications once while using a WAF and once without. The results show that the prototype tool, which uses a proxy as an oracle, detects more vulnerabilities than SqlMap, which relies only on the output of the application, for both applications and using both set-ups (with and without WAF). The only exception is the Hotel Reservation Service without a firewall where both tools find the same number (6) of vulnerabilities. Manual inspection showed that none of the vulnerabilities reported by either tool is a false positive. We also observe that the number of tries or test cases that the tool needs to execute before detecting the vulnerability (if a vulnerability is found) is significantly higher for SqlMap compared to the prototype tool (1,306.55 vs 6.3 and 566.77 vs 2). These results indicate that using a database proxy as an oracle for SQL injection testing could improve detection rates and could also enhance the efficiency of the testing process by detecting vulnerabilities faster.

The difference in the number of detected vulnerabilities when testing with and without a WAF can help us answer RQ2. As we expected, testing through a WAF is more challenging and both testing tools find fewer vulnerabilities in both applications. The only exception is the result of the prototype tool for the Hotel Reservation Service, where the tool found the same number of vulnerabilities with and without a firewall. SqlMap was unable to detect any vulnerabilities for both applications when using a firewall. We also noticed that when vulnerabilities are found, the number of tries needed to find the vulnerabilities also significantly increased (34 vs 2 and 28 vs 6.3). The prototype tool found three vulnerabilities when testing through the WAF for the SugarCRM application. Therefore, these three vulnerabilities are unprotected by the WAF and any debugging or fault repairing effort should be first focused on these three vulnerabilities since the risk of them being exploited is higher. Another conclusion we might draw from these results is that we need more sophisticated test generation techniques and oracles for SQL injection testing. SqlMap, a state-of-the-art-tool, was unable to find any vulnerabilities in both applications when using a WAF, while our prototype tool detected six in one application and three out of six in the other. A more sophisticated test generation technique might be able to detect vulnerabilities not found by either tool. Moreover, as hackers are continuously searching for new ways and attack patterns to penetrate WAFs and find security holes in applications, SQL injection testing should attempt to emulate these attackers and identify vulnerabilities before the attackers do.

As we noticed in the results that the prototype tool finds only three out of the six vulnerabilities in the SugarCRM application, we investigated the reasons and the difference, if any, between the detected and undetected vulnerabilities when using the WAF. Surprisingly, we found that the WAF blocks even the valid request when testing the operations that have the three undetected vulnerabilities. The reason that the valid requests are blocked for these operations is that some of their parameters are formatted as a series of numbers and letters separated by dashes. The rule set we used for ModSecurity (our WAF) includes a rule that blocks any request that contains more than five spacial characters (e.g., hash signs, quotes, dashes), which these input parameters trigger causing the request to be blocked. We expect such cases to not happen in practice in real systems: Security engineers would customise the configuration and rule set of the WAF to ensure that the normal operations and functionality of the application are not affected. This highlights the need for using real industrial case studies when evaluating tools and techniques to obtain more realistic results that reflect what happens in real systems and contexts. Such studies are unfortunately very rare in the research literature.

We also examined the test cases that successfully detected vulnerabilities when using our prototype tool. We found that these test cases changed the structure of the SQL statements they affected causing GreenSQL to flag them as SQL injection attacks. However, we also found that the resulting SQL statements were not executable. For example, one of the attack strings used that detected a vulnerability was `' UNION SELECT`. The vulnerable SQL statement was:

```
 $sql="Select * From hotelList where country ='".$country.'"';
```

The SQL statement after injecting the variable `$country` with the attack string `' UNION SELECT` would be:

```
Select * From hotelList where country =' ' UNION SELECT'
```

GreenSQL will detect this statement as an SQL injection attack since the structure of this statement differs from the previously learnt statements. However, the statement itself is not executable and would cause the database server to raise a syntax error when attempting to execute the statement. If the resulting SQL statement was syntactically correct and executable, we might be able to have more confidence in that the detected vulnerability is exploitable. If one of the test cases that detected the vulnerability was used by an attacker, he or she would not be able to gain any benefit from the attack. This suggests that we need to enhance the oracle to get more useful results that can help identify not just detectable vulnerabilities but also exploitable vulnerabilities and produce test cases that result in executable SQL statements that change the behaviour of the application. This can be done, for example, by improving the oracle by combining the database proxy with an additional oracle that checks the syntactical correctness of the resulting SQL statement.

5.5 Threats to Validity

This section discusses the threats to validity of our results in this study using the standard classification of threats [26]:

Internal Threats: The internal threats to validity in this study are related to generation of test cases and the stopping criterion of each approach when studying the effect of the test oracle. Both approaches start from a valid test case when testing each web service operation. We used the same initial test cases for both the prototype tool and SqlMap to avoid experimenter bias.

External Threats: The external threats are related to the choice of case study subjects, the SQL injection testing approaches and the ability to generalise results. Although we only used two systems in the case study, one of the two systems is used by real users as the number of downloads indicates. More experiments with different types of systems might be needed before being able to generalise results. Although we only used two approaches to generate test cases, these two approaches are representative of the state of the art in black-box testing, as the review of related work indicates.

Construct Threats: We used the number of detected vulnerabilities to measure effectiveness and used the number of test cases generated before a vulnerability is detected to measure efficiency. Detecting vulnerabilities is the goal of any SQL injection testing approach, therefore, the number of vulnerabilities seems like the most natural choice to measure effectiveness. The number of requests (or test cases) issued before detecting a vulnerability is a more reliable method of measuring efficiency since execution time might be effected by the environment and/or other processes performed by the CPU while running the experiments.

6 Conclusion

SQL injections are a significant and increasing threat to web applications. It is therefore highly important to test such applications in an effective manner to detect SQL injection vulnerabilities. In many situations, for example when the source code or adequate code analysis technologies are not available, one must resort to black-box testing. This paper examined the impact of Web Application Firewalls (WAFs) and database proxies on black-box SQL injection testing. We proposed using WAFs to prioritise fixing SQL injection vulnerabilities by testing the application with and without using a WAF and then prioritising fixing vulnerabilities that are not protected by the WAF. We also proposed using database proxies, which monitor the communications between the application and the database and flag any suspicious SQL statements, as an oracle for SQL injection testing.

We conducted a case study on two service oriented web applications where we compared the effectiveness and efficiency of two SQL injection tools: SqlMap, which is a state-of-the-art black-box testing tool that uses the output of the application as an oracle and a prototype tool we developed that uses a database proxy (GreenSQL) as an oracle. The results confirmed that using a database proxy increases the detection rates of SQL injection testing and also results in finding vulnerabilities with significantly lower numbers of test cases. A more detailed investigation of the test cases produced revealed that using database proxies helps in detecting more vulnerabilities but a more sophisticated oracle is needed to be able to reason about the vulnerabilities' exploitability, i.e., if an attacker would be able to gain any benefit from the vulnerability.

We also compared the results of the two testing tools when testing through a WAF (ModSecurity) and when testing the applications directly. The results showed that testing through the WAF is more challenging, causing our prototype tool to only detect 50 % of vulnerabilities for one application, while SqlMap detected vulnerabilities for neither application. These results have two implications: Testing through WAFs can be used to prioritise fixing vulnerabilities that are not protected by the WAF. On the other hand, the inability of SqlMap to detect any vulnerabilities when testing through the WAF, although some of those vulnerabilities were detectable by our prototype tool, suggests that we need to improve further both the test generation and oracles of black-box SQL injection testing.

Acknowledgment. This work is supported by the National Research Fund, Luxembourg (FNR/P10/03 and FNR 4800382).

References

1. Antunes, N., Vieira, M.: Detecting SQL injection vulnerabilities in web services. In: Proceedings of the 4th Latin-American Symposium on Dependable Computing (LADC '09), pp. 17–24 (2009)
2. Apache-scalp: Apache log analyzer for security (2008). https://code.google.com/p/apache-scalp
3. Bau, J., Bursztein, E., Gupta, D., Mitchell, J.: State of the art: automated black-box web application vulnerability testing. In: Proceedings of the 2010 IEEE Symposium on Security and Privacy (SP '10), pp. 332–345 (2010)
4. Beery, T., Niv, N.: Web application attack report (2011)
5. Christey, S., Martin, R.A.: Vulnerability type distributions in CVE (2007). http://cwe.mitre.org
6. Ciampa, A., Visaggio, C.A., Di Penta, M.: A heuristic-based approach for detecting SQL-injection vulnerabilities in web applications. In: Proceedings of the ICSE Workshop on Software Engineering for Secure Systems (SESS '10), pp. 43–49 (2010)
7. Coffey, J., White, L., Wilde, N., Simmons, S.: Locating software features in a SOA composite application. In: Proceedings of the 8th IEEE European Conference on Web Services (ECOWS '10), pp. 99–106 (2010)
8. Damele, B., Guimaraes, A., Stampar, M.: Sqlmap (2013). http://sqlmap.org/
9. Doupé, A., Cova, M., Vigna, G.: Why Johnny can't pentest: an analysis of black-box web vulnerability scanners. In: Kreibich, C., Jahnke, M. (eds.) DIMVA 2010. LNCS, vol. 6201, pp. 111–131. Springer, Heidelberg (2010)

10. Elia, I.A., Fonseca, J., Vieira, M.: Comparing SQL injection detection tools using attack injection: an experimental study. In: Proceedings of the IEEE 21st International Symposium on Software Reliability Engineering (ISSRE '10), pp. 289–298 (2010)
11. Fossi, M., Johnson, E.: Symantec global internet security threat report, vol. xiv (2009)
12. Fu, X., Qian, K.: SAFELI: SQL injection scanner using symbolic execution. In: Proceedings of the workshop on Testing, Analysis, and Verification of Web Services and Applications (TAV-WEB '08), pp. 34–39 (2008)
13. GreenSQL LTD: Greensql (2013). http://www.greensql.com
14. Halfond, W.G., Anand, S., Orso, A.: Precise interface identification to improve testing and analysis of web applications. In: Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA '09), pp. 285–296 (2009)
15. Hanna, S., Shin, R., Akhawe, D., Boehm, A., Saxena, P., Song, D.: The emperors new apis: on the (in) secure usage of new client-side primitives. In: Proceedings of the Web, vol. 2 (2010)
16. Holler, C., Herzig, K., Zeller, A.: Fuzzing with code fragments. In: Proceedings of the 21st Usenix Security Symposium (2012)
17. Huang, Y.-W., Huang, S.-K., Lin, T.-P., Tsai, C.-H.: Web application security assessment by fault injection and behavior monitoring. In: Proceedings of the 12th International Conference on World Wide Web (WWW '03), pp. 148–159 (2003)
18. Khoury, N., Zavarsky, P., Lindskog, D., Ruhl, R.: Testing and assessing web vulnerability scanners for persistent SQL injection attacks. In: Proceedings of the 1st International Workshop on Security and Privacy Preserving in e-Societies (SeceS '11), pp. 12–18 (2011)
19. Kieyzun, A., Guo, P.J., Jayaraman, K., Ernst, M.D.: Automatic creation of SQL injection and cross-site scripting attacks. In: Proceedings of the 31st International Conference on Software Engineering (ICSE '09), pp. 199–209 (2009)
20. PCI Security Standards Council: Pci data security standard (PCI DSS) (2013). https://www.pcisecuritystandards.org
21. Roesch, M.: Snort - lightweight intrusion detection for networks. In: Proceedings of the 13th USENIX Conference on System Administration, pp. 229–238 (1999)
22. Ryck, P.D., Desmet, L., Philippaerts, P., Piessens, F.: A security analysis of next generation web standards (2011)
23. Shar, L.K., Tan, H.B.K.: Mining input sanitization patterns for predicting SQL injection and cross site scripting vulnerabilities. In: Proceedings of the 34th International Conference on Software Engineering (ICSE NIER '12), pp. 1293–1296 (2012)
24. The Open Web Application Security Project (OWASP): Testing for SQL injection (owasp-dv-005) (2013). http://www.owasp.org
25. Vieira, M., Antunes, N., Madeira, H.: Using web security scanners to detect vulnerabilities in web services. In: Proceedings of the IEEE/IFIP International Conference on Dependable Systems & Networks (DSN'09), pp. 566–571 (2009)
26. Wohlin, C., Runeson, P., Host, M., Ohlsson, M., Regnell, B., Wesslen, A.: The Experimentation in Software Engineering - An Introduction. Kluwer, Dordrecht (2000)