Testing and comparing web vulnerability scanning tools for SQL injection and XSS attacks

José Fonseca
CISUC - Polithecnic Institute of Guarda
6300 Guarda
Portugal
josefonseca@ipg.pt

Marco Vieira, Henrique Madeira
DEI/CISUC - University of Coimbra
3030 Coimbra
Portugal
{mvieira, henrique}@dei.uc.pt

Abstract

Web applications are typically developed with hard time constraints and are often deployed with security vulnerabilities. Automatic web vulnerability scanners can help to locate these vulnerabilities and are popular tools among developers of web applications. Their purpose is to stress the application from the attacker's point of view by issuing a huge amount of interaction within it. Two of the most widely spread and dangerous vulnerabilities in web applications are SQL injection and Cross Site Scripting (XSS), because of the damage they may cause to the victim business. Trusting the results of web vulnerability scanning tools is of utmost importance. Without a clear idea on the coverage and false positive rate of these tools, it is difficult to judge the relevance of the results they provide. Furthermore, it is difficult, if not impossible, to compare key figures of merit of web vulnerability scanners. In this paper we propose a method to evaluate and benchmark automatic web vulnerability scanners using software fault injection techniques. The most common types of software faults are injected in the web application code which is then checked by the scanners. The results are compared by analyzing coverage of vulnerability detection and false positives. Three leading commercial scanning tools are evaluated and the results show that in general the coverage is low and the percentage of false positives is very high.

1. Introduction

Web applications are extremely popular today. Nearly all information systems and business applications (e-commerce, banking, transportation, web mail, blogs, etc) are now built as web-based database applications. They are so exposed to attacks that any existing security vulnerability will most probably be uncovered and exploited, which may have a highly negative impact on users. Automatic web vulnerability scanners are often used by web application developers and system administrators to test web applications against vulnerabilities. Therefore, trusting the results of web vulnerability scanners is essential. To what extent can one trust the verdict delivered by web vulnerability scanners, especially when the tool report suggests that there are no vulnerabilities in the web application? The answer to this question is the focal point of this paper.

Traditional security mechanisms like network firewalls, intrusion detection systems (IDS), and use of encryption can protect the network but cannot mitigate attacks targeting web applications, even assuming that key infrastructure components such as web servers and database management systems (DBMS) are fully secure. Hence, hackers are moving their focus from network to web applications where poor programming code represents a major risk. This can be confirmed by numerous vulnerability reports available in specialized sites like www.securityfocus.com, www.ntbugtraq.com, www.kb.cert.org/vuls, etc.

This year the Open Web Application Security Project released its ten most critical web application security vulnerabilities [1] based on data provided by Mitre [2]. This report ranked XSS as the most critical vulnerability, followed by Injection Flaws, particularly SQL injection.

Computer Security Institute/FBI concluded in a survey [3] that defacement of web sites is a problem for many organizations, as 92% of the respondents reported more than 10 web site incidents. Another study provided by the Gartner Group reveals that 75% of the attacks are on web based applications [4]. An Acunetix audit result says "on average 70% of websites are at serious and immediate risk of being hacked... and... 91% of these websites contained some form of website vulnerability, ranging from the more serious ones such as SQL Injection and Cross Site Scripting (XSS)…" [6]. These attacks basically take advantage of improper coded applications due to unchecked input fields at user interface. This allows the attacker to change the SQL commands that are sent to the database (SQL Injection) or through the input of HTML and a scripting language (XSS). The popularity of these exploitations is due to: the easiness of finding and exploiting such vulnerabilities [2]; the importance of the assets they can disclosure; and the level of damage they may inflict. These allow attackers to access unauthorized data (read, insert, change or delete), gain access to privileged database accounts, impersonate another user, mimicry web applications, deface web pages, get access to the web server, etc.

To prevent this scenario developers are encouraged to follow the best coding practices, perform security reviews of the code and regular auditing, to use code vulnerability analyzers, etc. However, web application developers normally focus on application functionalities and on satisfying the user's requirements due to time constraints, and easily neglect security aspects. Even the common widely used Rapid Application Development environments produce code with vulnerabilities.

Web vulnerability scanners are often regarded as an easy way to test the security of web applications, including critical vulnerabilities such as SQL injection and XSS. The approach followed in this paper consists of injecting software faults into a web application code and checking if web vulnerability scanners can detect the potential vulnerabilities created by the injected faults. The possible creation of vulnerabilities is confirmed manually in order to get accurate measures of the detection coverage and false positives. The software faults injected represent the most common types of software faults found in a field study [5]. These have been adapted for the web application environment.

Fault injection techniques are quite common for assessment of critical fault tolerant systems. Most of the techniques inject hardware faults (e.g., [7]) or emulate the injection of hardware faults by software (e.g., [8]). The injection of realistic software faults (i.e., program defects or bugs) is relatively new [9, 10]. In this paper we use the Generic Software Fault Injection Technique (G-SWFIT) [5] to inject common programmer bugs in the web application code.

The structure of the paper is as follows. Section 2 presents some background on automatic web vulnerability scanners. Section 3 describes the proposed approach to test and compare web vulnerability scanners. Section 4 presents the experiments and discusses the results and Section 5 concludes the paper and describes future work.

2. Detection of vulnerabilities in web applications

There are two main approaches to test web applications for vulnerabilities: "white box" and "black box". The "white box" approach consists of the analysis of the source code of the web application. This can be done manually or by using code analysis tools like FORTIFY [11], Ounce [12], Pixy [13], etc. To detect SQL injection the static analyzer tool uses the web application code to follow all the possible paths and the changes it may go through due to the manipulation process of the SQL query text and finally parses the result. Exhaustive source code analysis may not find all security flaws because of the complexity of the code. In these situations it is preferable to use the "black box" approach. In this approach the scanner does not know the internals of the web application and it uses fuzzing techniques over the web HTTP requests. It provides an automatic way to search for vulnerabilities avoiding the repetitive and tedious task of doing hundreds or even thousands of tests by hand for each vulnerability type. This technique is called penetration testing and is actually a form of robustness testing, as the tool submits nonsense or malicious values to the web application evaluating its response to see if the penetration attempts were successful. According to the survey presented in [3], penetration testing is the second most used technique to evaluate the effectiveness of security, being used by 66% of the respondents. That is why the methodology proposed in this paper adopts the "black box" testing approach using web vulnerability scanners.

There are many commercial web vulnerability scanners such as Acunetix Web Vulnerability Scanner [14], Spi Dynamics Webinspect [15], Watchfire AppScan [16], Buyservers Falcove [17], N-Stalker Web Application Security Scanner [18], and Cenzic Hailstrom [19]. Examples of free web vulnerability scanners include Gamja [20] and BrupSuite [21], but they are usually limited scripting tools not fully automatic as their commercial equivalent.

These scanners include normally three main stages: configuration, crawling, and scanning. The configuration stage includes the definition of the Uniform Resource Locator (URL) of the web application and the setup of parameters.

In the crawling stage the vulnerability scanner produces a map of the internal structure of the web application. This stage is of utmost importance because failing to discover some pages of the application will prevent their testing (in the subsequent scanning stage). The scanner calls the first web page and then examines its code searching for links. Each link found is requested and this procedure is executed over and over until no more links or pages can be found.

The scanning stage is where the automated penetration test is performed against the web application by simulating a browser user clicking on links and filling in form fields. During this stage thousands of tests are executed. Malformed requests are also sent in order to learn the error responses. The requests and the responses are recorded and analyzed using vulnerability policies. The responses are also validated using data collected during the crawling stage. During this stage new links are frequently discovered and when this happens they are added to the result of the crawler in order to be also scanned for vulnerabilities.

After the scanning stage the results are shown to the user and they may be saved for later analysis. Most scanners also show some generic information about the vulnerabilities discovered, including how to avoid or correct them. Besides the graphical user interface, most scanners also have a command line application with several parameters aimed for automation by using batch jobs.

Scanners use the layout engine of one internet browser to process the responses of the web server. There are several layout engines available, like Gecko from Mozilla, WebKit from Safari, Presto from Opera, but the scanners usually use the Trident from Internet Explorer. The layout engines interpret the HTML code differently and do not completely support its related standards [22]. Some vulnerabilities affect only a specific browser or version, usually due to the relaxed way the layout engine treats the HTML code.

Scanners also have a collection of signatures of known vulnerabilities of different versions of web servers, operating system and also of some network configurations. These signatures are updated regularly as new vulnerabilities are discovered. They also have a pre-defined set of tests of some generic types of vulnerabilities like SQL injection and XSS. In the search for vulnerabilities like XSS and SQL injection, the scanners execute lots of pattern variations adapted to the specific test in order to discover the vulnerability and to verify if it is not a false positive. The tests for these kinds of vulnerabilities, including both the sequences of input values and the way to detect success or failure, are quite different from scanner to scanner, so the results obtained by different tools may vary a lot (this is actually one of the reasons why it is so important to have means to compare different scanners).

3. Web vulnerability scanners benchmarking approach

The proposed approach to evaluate the web application vulnerability scanners consists of injecting realistic software faults (one fault at each time) in the code of a given web application. The results of the various scanners are compared evaluating the efficiency in identifying the potential vulnerabilities created by the injected fault. In other words, if web vulnerability scanners are supposed to detect vulnerabilities (which are caused by residual software faults in the web application code), then our idea consists of providing the scanners with the input they are supposed to handle, which is a web code with software faults and possible vulnerabilities originated by such faults.

The process of injecting faults in the web application, running the vulnerability scanners, and analyzing and comparing the results is completely automatic. However, in these first experiments we decided to inspect manually the potential vulnerabilities caused by each injected fault in order to have precise control on the experiment results and get precise measures on the percentage of vulnerabilities and false positives detected by each tool.

The following subsections discuss the software fault injection process and describe the proposed benchmarking procedure in detail.

3.1. Software fault injection in web applications

The variety of mistakes (i.e., software bugs) found in a deployed code tends to be enormous [23], which makes the exhaustive classification of software faults a cumbersome task. Nevertheless, field studies show that the most frequent types of software faults belong to a small group [5, 9]. In other words, there are a small number of software fault types that account for the majority of faults found in a field, while there is a huge variety of relatively rare types of faults [5].

The software fault injection technique used in this paper is the G-SWFIT. This technique just focuses on the emulation of the most frequent types of faults. It is based on a set of fault injection operators that reproduce directly in the target executable code the instruction sequences that represent most common types of high-level software faults. These fault injection operators resulted from a field study that analyzed and classified more than 600 real software faults discovered in several programs, identifying the most common (the “top-N”) types of software faults

[5]. Table 1 shows the 12 most frequent types of faults found in [5] that we use in the present paper. The ODC classes are fault classes defined according to the IBM's ODC classification [23].

The locations in the target code where the injection is performed are selected by the G-SWFIT tool in order to assure that a given fault type is only injected in a code location where that kind of fault could realistically exist. For example, MIFS fault type in Table 1 can only be injected in target code locations that represent an IF structure. Furthermore, in [5] are defined a set of restrictions (based on the field observations) used by the G-SWFIT tool to increase the realism of the injected fault.

The original G-SWFIT operators were not defined with a web application code in mind, as the field study reported in [5] mainly addressed programs written in C. Although the G-SWFIT fault operators were also evaluated for other languages (see [5] for details), none of them were typical programming languages used for the development of web applications (e.g., PHP, ASP, Java, .NET). Thus, small adaptations in the fault operators proposed in [5] had to be introduced to use them for our purposes. Most of the changes are trivial adaptations such as the one used for the "*MVI*" operator. As in PHP there is no need for variable initialization so we applied this operator in the first assignment of a variable (and not in the initialization). Another small change is in the "*MIA*" operator where we use it even in the situation where there is one "else" but it is closely related to the "if", like the display of an error message. The biggest change was in the "*MFC*" operator. In web application programming there are normally lots of functions that process a parameter and returns the same data type as the parameter or a data type that can be used correctly by the program. For example in PHP code we can have:

```
<? echo 'test.php?id='.urlencode($id); ?>
```

where the "urlencode" function encodes the string variable "$id" to be passed as a GET parameter in the URL. If the developer forgets to use the "`urlencode($id)`" therefore using only the "`$id`" variable, the code can still be interpreted without any problem by the web server. So it is feasible that the software developer may forget to use this function and pass the "$id" directly as the GET parameter. However according to [5] we could not insert this kind of fault because it fails to follow the restriction of the "*MFC*" operator. According to [5] this operator should be applied only when the return value of the function is not being used by any of the subsequent instructions. To overcome this situation we relaxed the restriction defined in [5] and created a new operator named "*Missing function call extended*". All the other fault operators were used as defined in [5] (with minor adjustments in some classes, as mentioned above).

3.2. Testing procedure

The injection of the software faults (one fault at a time) consists of two stages:

- In the first stage the code of the target web application is examined in order to identify all the points where each type of fault can be injected, resulting in a list of possible faults. When the list of faults is very large (because the application code is extensive, resulting in lots of possible locations for each fault type), only a percentage of the fault locations is used, keeping the relative percentages shown in Table 1.
- The second stage comprises the injection of each fault, which corresponds to the insertion of the code change (defined by the fault operator) in the web application. After injecting each fault, the web application is scanned by the tools under evaluation to compare their results.

The proposed testing procedure needs two computers connected to an Ethernet network. One computer acts as a server executing the functions of a web server, an application server and a database server. The other computer acts as a client with a web

Table 1 – Most frequent fault types found in [5]

Fault types	Description	% of total observed in field study	ODC classes
MIFS	Missing "If (*cond*) { statement(s) }"	9.96 %	Algorithm
MFC	Missing function call	8.64 %	Algorithm
MLAC	Missing "AND EXPR" in expression used as branch condition	7.89 %	Checking
MIA	Missing "if (*cond*)" surrounding statement(s)	4.32 %	Checking
MLPC	Missing small and localized part of the algorithm	3.19 %	Algorithm
MVAE	Missing variable assignment using an expression	3.00 %	Assignment
WLEC	Wrong logical expression used as branch condition	3.00 %	Checking
WVAV	Wrong value assigned to a value	2.44 %	Assignment
MVI	Missing variable initialization	2.25 %	Assignment
MVAV	Missing variable assignment using a value	2.25 %	Assignment
WAEP	Wrong arithmetic expression used in parameter of function call	2.25 %	Interface
WPFV	Wrong variable used in parameter of function call	1.50 %	Interface
	Total faults coverage	**50.69 %**	

browser. It is in this computer where the vulnerability scanner is executed.

The experiments need some operations executed in the server computer and some in the client computer, in synchronism. Because of the large number of tests and the long duration of the whole process a Java software tool was developed to automate the procedure. This Java tool is deployed in the client computer and is able to communicate with the server computer in order to be able to automatically execute all the procedures needed by the tests.

The experiments start with a "gold run" where the web application is tested once by each vulnerability scanner without any faults injected. The web application may already have some vulnerabilities and this run will be able to find most of them.

After the "gold run" the Java tool reads the file with fault definitions (set of faults to inject identified in the first fault injection stage) that will be used in the tests. Then, for each fault, the following procedure is executed:

1) Every test starts with a restoration of the initial setup: the web server is restarted; the database is restored; and the website files are copied from a clean backup.

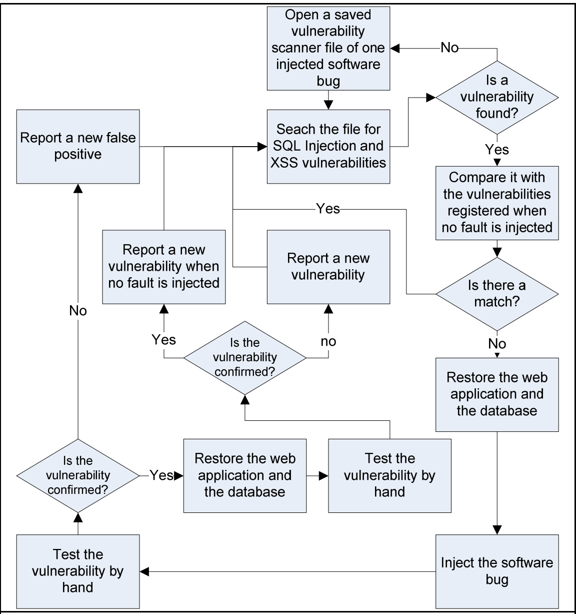
2) The next fault is injected into the web application.

3) The scanner application is started and at the end the results are saved into a file. The file name includes a reference to the delta file, the web application file and the type of fault injected. The Java tool monitors the scanner application in order to detect when its execution stops before continuing the next test.

4) This procedure is repeated from 1) to 3) until all the faults are injected.

5) This procedure (from steps 1 to 4) is also repeated for all the web vulnerability scanners.

After all tests have been performed, every file resulting from the execution of the scanners is manually analyzed using the algorithm presented in Figure 1. In these experiments we are only interested in the XSS and SQL Injection vulnerabilities, so when the scanner reports other types of vulnerabilities they are ignored. All the reported vulnerabilities are checked for false positives. It is also verified if the vulnerability is derived from the fault injected or if it is a vulnerability that was already present in the application and has not been detected in the "gold run". The vulnerabilities are also verified manually to confirm that they are unique and not the same vulnerability tested in a different way. This happens when the same vulnerable code is executed when the web application code is called from different places. For instance, when we press the "Insert" button or the



Open a saved vulnerability scanner file of one injected software bug
No
Is a vulnerability found?
Report a new false positive
Seach the file for SQL Injection and XSS vulnerabilities
Yes
Compare it with the vulnerabilities registered when no fault is injected
Yes
Report a new vulnerability when no fault is injected
Report a new vulnerability
Is there a match?
No
No
Yes
Is the vulnerability confirmed?
no
Restore the web application and the database
Is the vulnerability confirmed?
Yes
Restore the web application and the database
Test the vulnerability by hand
Test the vulnerability by hand
Inject the software bug

Figure 1: Algorithm applied to the scanner generated files.

"Update" button in a FORM they may execute some common code. If the vulnerability is in the common code they will be triggering the same vulnerability and it should only be accounted only once.

4. Case study experiments and discussion

For the evaluation experiments we used LAMP (Linux, Apache, Mysql and PHP) web applications. The server runs Linux and the web server is Apache. This server hosts a PHP developed web application using a Mysql database. This topology of operating system and software was chosen because it represents one of the most used technologies to build custom web applications nowadays. It is also responsible for a large number of SQL injection and XSS security vulnerabilities, which are our target vulnerabilities.

We used three commercial web application vulnerability scanners, that we named Scanner 1, Scanner 2, and Scanner 3. We have decided to keep the brand and the versions of the web vulnerability scanners anonymous to assure neutrality and because commercial licenses do not allow in general the publication of tool evaluation results.

In order to obtain a more complete evaluation of the three scanners, we decided to test two web applications. One of the target web applications is custom made and is called MyReferences. It is mainly used to manage personal reference information. It allows the storage of pdf documents, and some information like their title, authors and year of publication. The underlined database used has

Table 2a – MyReferences experimental results.

		Scanner 1		Scanner 2		Scanner 3		sum of the distinct vulnerabilities found by scanners			
Fault Types	**# Faults**	**XSS**	**SQL Inject.**	**XSS**	**SQL Inject.**	**XSS**	**SQL Inject.**	**XSS**	**SQL Inject.**	**#**	**%**
No Fault Injected	0	7	0	1	1	11	1	12	2	14	
MIFS	23	1	1	0	0	1	1	1	1	2	**9%**
MFC	26	0	0	0	0	0	0	0	0	0	**0%**
MFC extended	71	8	5	2	16	6	36	20	39	59	**83%**
MLAC	48	2	0	0	0	0	0	2	0	2	**4%**
MIA	55	4	7	2	1	1	8	5	10	15	**27%**
MLPC	97	0	0	0	0	0	0	0	0	0	**0%**
MVAE	80	0	0	0	0	0	0	0	0	0	**0%**
WLEC	76	3	7	3	3	0	8	7	12	19	**25%**
WVAV	13	0	0	0	0	0	0	0	0	0	**0%**
MVI	8	0	0	0	0	0	0	0	0	0	**0%**
MVAV	13	0	0	0	0	0	0	0	0	0	**0%**
WAEP	1	0	0	0	0	0	0	0	0	0	**0%**
WPFV	148	0	13	0	0	0	12	2	19	21	**14%**
Total (injected)	**659**	**25**	**33**	**8**	**21**	**19**	**66**	**49**	**83**	**118**	**18%**

currently 114 publications stored and 311 authors. The web application code consists of 12 PHP files and an overall of 1,436 lines of code. The second web application is the Online BooksStore (http://www.gotocode.com/apps.asp?app_id=3). It is a fully functional and ready to use online store generated with the Rapid Web Application Development Framework CodeCharge (http://www.yessoftware.com/products/product_detail.php?product_id=1). This application is composed of 29 PHP files with a total of 9,437 lines of code.

All the 12 most frequent types of faults found in Table 1 were used in the experiments for the MyReferences web application. Every filename of the web application was analyzed looking for locations of every fault type. In the MyReferences 659 faults were injected (Table 2a).

Due to time constraints during the testing with the BookStore web application, we just have preliminary result of the experiments of the injection of 3 types of faults and only two vulnerability scanner applications. Because of its size we found 1,322 fault locations, thus we applied the percentages of total observed in field study shown in Table 1 resulting in the injection of 327 faults in the web application (Table 2b).

The faults injected leaded to application bugs and application malfunctioning, but they also produced a considerable amount of security vulnerabilities (18% for the MyReferences web application and 4% for the BookStore application). Note that some injected bugs produced more than one type of vulnerabilities (XSS and SQL Injection) and some produced more than one vulnerability of the same type. The number of vulnerabilities found without any fault injected in the BookStore application was 29. We consider this a very high number because these intrinsic vulnerabilities masquerade the discovery of new vulnerabilities leaving less code to inject them.

As we can see, from the 12 fault types only 6 produced vulnerabilities. They are the *MIFS* the *MFC extended*, the *MLAC*, the *MIA*, the *WLEC* and the *WPFV*. The vulnerabilities generated by every fault type were both XSS and SQL Injection.

The distribution of the two types of vulnerabilities is shown in Table 3a and Table 3b. Fault injection produced more than the double of SQL Injection type than the XSS type for the MyReferences and almost the opposite for the BookStore showing that there is no pattern regularity in this segmentation of the results.

Table 3a – MyReferences type of vulnerabilities

	SQL Injection	**XSS**
#	81	37
%	69%	31%

Table 3b – BookStore type of vulnerabilities

	SQL Injection	**XSS**
#	5	8
%	38%	62%

In terms of the way the vulnerability may be

Table 2b – BookStore experimental results (not complete results yet)

		Scanner 1		Scanner 3		sum of the distinct vulnerabilities found by scanners			
Fault Types	**# Faults**	**XSS**	**SQL Injec.**	**XSS**	**SQL Inject.**	**XSS**	**SQL Inject.**	**#**	**%**
No Fault Injected	**0**	**12**	**0**	**22**	**1**	**27**	**1**	**29**	
MIFS	**120**	**4**	**0**	**4**	**0**	**4**	**0**	**4**	**3%**
MFC	**103**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0%**
MFC extended	**104**	**3**	**3**	**3**	**4**	**4**	**5**	**9**	**9%**
Total (injected)	**327**	**19**	**3**	**29**	**5**	**35**	**6**	**42**	**4%**

exploited there are much more vulnerabilities that are exploited through the HTTP GET submission method than through the HTTP POST submission method in both applications (Table 4a, Table 4b). Although the HTTP GET can be exploited more easily by an attacker because all it needs is to change the URL we believe that these values may change according to the submission methods used by the web application.

Table 4a – MyReferences HTTP submission methods

	GET	POST
#	71	47
%	60%	40%

Table 4b – BookStore HTTP submission methods

	GET	POST
#	9	4
%	69%	31%

All the scanners have detected some vulnerabilities that none of the others have. To analyze the scanners coverage a manual inspection was performed by a human tester examining both the PHP code and the browser results. This hand scan detected 17 vulnerabilities that have not been detected by neither of the automatic vulnerability scanners, corresponding to 9% of all the vulnerabilities found (Figure 2a, Figure 2b, Figure 2c). For the BookStore application a complete hand scan was not done due to time constraints, however some quick tests show the existence of some second order vulnerabilities that were not detected by the scanners, which confirms the trend observed in the MyReferences experiments.

The intersection area of the circles (Figure 2a, Figure 2b, Figure 2c) represent vulnerabilities detected by more than one scanner (the number of vulnerabilities detected are shown). As we can see, the circle representing manual scan does not intersect with the other circles, which means that the vulnerabilities detected by manual inspection were not detected by any of the tools evaluated. It is also worth noting that the radius of each circle is proportional to the number of vulnerabilities detected, providing a comparative graphic image of the coverage of each tool. The observation of Figure 2c clearly shows that scanner 3 is the best one concerning coverage of vulnerability detection, followed by scanner 1 and scanner 2. However, we can also see that the best scanner for SQL Injection is not the best for XSS (Figure 2b).

While some undetected vulnerabilities should have been detected by the vulnerability scanners there are others that would be difficult for a tool using the "black box" approach, such as second order vulnerabilities. In the first stage of this vulnerability type the attacker changes something that would not be noticed immediately. This change would only perform its malicious action later on (second stage) when some condition is encountered. For example, in the first stage the attacker inserts in a database field a malicious string containing an XSS exploit. By introducing this string nothing strange happens in the web application or in the database. The second stage may be the activation of the code injected in stage one by a bug that is triggered when the malicious string is displayed by the browser. Another difficulty for the scanners is when the exploit needs some specific tokens to be present. These tokens may be the right number of parenthesis in a SQL Injection attempt, or some precise HTML code in an XSS attack. Although the scanners have some fuzzy variations of tests, these will hardly cover all the possible combinations.

Although the scanners have found most of the vulnerabilities they also have detected many false positives, as shown in Table 5a and Table 5b. As we expected the false positive rate tends to be directly proportional to the capacity to detect vulnerabilities.

Table 5a – MyReferences false positives

	Scanner 1	Scanner 2	Scanner 3
#	13	43	45
%	20%	62%	38%

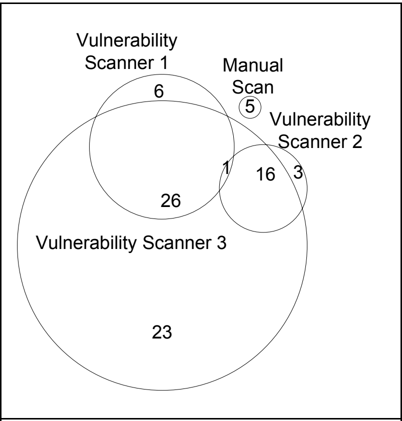


Figure 2a: MyReferences SQL Injection coverage

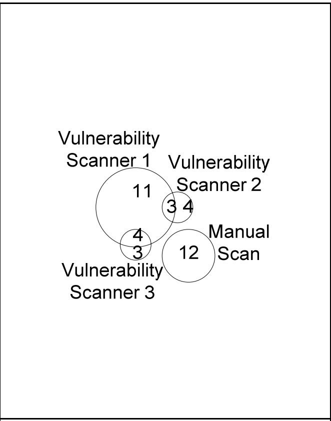


Figure 2b: MyReferences XSS coverage

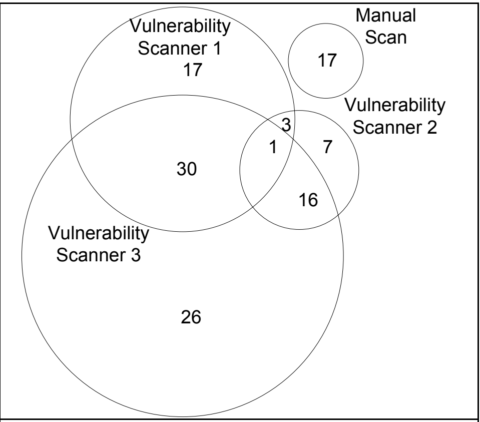


Figure 2c: MyReferences total coverage

Table 5b – BookStore false positives

	Scanner 1	Scanner 2	Scanner 3
#	6		36
%	38%		77%

Moreover there is a characterization of false positives types found in the MyReferences application. One of them is an error issued by the web application in normal execution due to the fault injected. This error message was found in 10 cases of the scanner 1, in 43 cases of the scanner 3, and in 40 cases of the scanner 2. In the penetration test the same error was shown and that triggered the scanner. The other three remaining cases of false positives found by scanner 1 and the two remaining by scanner 3 were not possible to be reproduced. The three remaining false positives found by scanner 2 were curiously triggered by the title of a paper about SQL Injection, stored in the web application backend database.

The analysis of the false positives of the BookStore application found 7 cases of an erroneous logout of the web application. In 3 other cases the attack can not be reproduced and in the other cases the false positive is due to error messages triggered by the fault injected.

5. Conclusion

In this paper we propose an approach to evaluate and compare web application vulnerability scanners. It is based on the injection of realistic software faults in web applications in order to compare the efficiency of the different tools in the detection of the possible vulnerabilities caused by the injected bugs. The results of the evaluation of three leading web application vulnerability scanners show that different scanners produce quite different results and that all of them leave a considerable percentage of vulnerabilities undetected. The percentage of false positives is very high, ranging from 20% to 77% in the experiments performed. The results obtained also show that the proposed approach allows easy comparison of coverage and false positives of the web vulnerability scanners. In addition to the evaluation and comparison of vulnerability scanners, the proposed approach also can be used to improve the quality of vulnerability scanners, as it easily shows their limitations. For some critical web applications several scanners should be used and a hand scan should not be discarded from the process.

For future work we intend to apply this benchmark procedure to other web applications to better understand the relationship between software faults and vulnerabilities. Knowing what kind of programming mistakes usually lead to security vulnerabilities can be an important tool to help in the detection and prevention of a security flaw. We also want to evaluate different configurations of the same scanner and study the association of scanners to cover a wider range of XSS and SQL Injection vulnerabilities.

6. References

[1] OWASP Foundation, 2007, http://www.owasp.org/index.php/Top_10_2007

[2] http://cwe.mitre.org/documents/vuln-trends.html

[3] L. Gordon, M. Loeb, W. Lucyshyn, R. Richardson, “Computer crime and security survey”, Computer Security Institute, 2006.

[4] http://www.gartner.com/

[5] J. Durães, H. Madeira, “Emulation of Software Faults: a Field Data Study and a Practical Approach”, Transactions on Software Engeneering, 2006.

[6] Acunetix Ltd, February 12, 2007, http://www.acunetix.com/news/security-audit-results.htm

[7] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E Martins, D. Powell, “Fault Injection for Dependability Validation — A Methodology and Some Applications”, Trans. on Software Engineering, 1990.

[8] J. Carreira, H. Madeira, J. Silva, "Xception: Software Fault Injection and Monitorintg in Processor Functional Units". IEEE Trans. on Software Engineering, vol. 24, 1998.

[9] J. Christmansson, R. Chillarege, “Generation of an Error Set that Emulates Software Faults”, Fault Tolerant Computing Symposium, 1996.

[10] J. Durães, H. Madeira, “Definition of Software Fault Emulation Operators: A Field Data Study”, The International Conference on Dependable Systems and Networks, 2003.

[11] FORTIFY, 2007, http://www.fortifysoftware.com/

[12] http://www.ouncelabs.com/

[13] http://pixybox.seclab.tuwien.ac.at/pixy/download.php

[14] Acunetix Ltd, Web Vulnerability Scanner, 2007, http://www.acunetix.com/vulnerability-scanner/

[15] SPI Dynamics, WebInspect, 2007, http://www.SpiDynamics.com/products/webinspect/index.html

[16] Watchfire Corporation, AppScan, 2007, http://www.watchfire.com/

[17] BuyServers Ltd., Falcove, 2007, http://www.buyservers.net/falcove.htm

[18] N-Stalker, 2007, http://www.nstalker.com/

[19] Cenzic, 2007, http://www.cenzic.com

[20] Gamja, 2007, http://lastlog.com/p4ssion/

[21] BrupSuite, 2007, http://portswigger.net/suite/

[22] David Hammond, 2007, http://www.webdevout.net/browser-support-summary

[23] R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. Moebus, B. Ray, M. Wong, “Orthogonal Defect Classification – A Concept for In-Process Measurement”, Transactions on Software Engineering, 1992.

[24] IBM Watson Research Center, http://www.watson.ibm.com/index.shtml