Using Vulnerability Injection to Improve Web Security

José Fonseca[1], Francesca Matarese[2]

[1] DEI/CISUC, University of Coimbra /
Polytechnic Institute of Guarda, 3030-290 Coimbra, Portugal
josefonseca@ipg.pt
[2] SESM Scarl, Giugliano in Campania, Italy
fmatarese@cesm.it

Abstract. This chapter presents a methodology to evaluate and benchmark web application vulnerability scanners using software fault injection techniques. The most common software faults are injected in the web application source code, which is then checked by the scanners. Using this procedure, we evaluated three leading commercial scanners, which are often regarded as an easy way to test the security of web applications, including critical vulnerabilities such as XSS and SQL Injection. Our idea consists of providing the scanners with the input they are supposed to handle, which is a web application with software faults and possible vulnerabilities originated by such faults. The results of the scanners are compared evaluating the efficiency in identifying the potential vulnerabilities created by the injected fault, their coverage of vulnerability detection and false positives. However, the results show that the coverage of these tools is low and the percentage of false positives is very high.

1 Introduction

The goal of a security program is to choose and implement cost effective countermeasures that mitigate the vulnerabilities that will most likely lead to loss. This chapter discusses how Vulnerability Management is one of the few counter-measures easily justified by its ability to optimize risk.

One of the most difficult issues security managers have is justifying how they spend their limited budgets. For the most part, information security budgets are determined by percentages of the overall IT budget. This implies that security is basically a “tax” on IT, as opposed to providing value back to the organization. The fact is that security can provide value to the organization, if there is a discussion of risk with regard to IT, as much as there is a discussion of risk with regard to all other business processes. Calculating a return on investment for a security countermeasure is extremely difficult as you rarely have the ability to calculate the savings from the losses you prevented.
However, if you start to consider that Security is actually Risk Management, you can start determining the best countermeasures to proactively and cost effectively mitigate your losses. By determining the vulnerabilities that are most likely to create loss, you can then compare the potential losses against the cost of the countermeasure. This

allows you to make an appropriate business decision as to justifying and allocating a security budget. More importantly, if you can make such a business decision, you can justify increasing security budgets for additional countermeasures. The key is to be able to specifically identify an area of potential loss, and identify a security countermeasure that cost effectively mitigates that loss [Winkler10].

Vulnerability injection is an innovative technique that can effectively increase vulnerability management by identifying the countermeasures that are really needed. Traditional countermeasures 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.

The Open Web Application Security Project released its ten most critical web application security vulnerabilities [OWASP10] based on data provided by Mitre Corp. [MITRE12]. This report ranked XSS as the most critical vulnerability, followed by Injection Flaws, particularly SQL injection.

Computer Security Institute/FBI concluded in a survey [Gordon06] that defacement of web sites is a problem for many organizations, as 92% of the respondents reported more than 10 web site incidents. 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)…" [Acunetix12]. 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 high risk of these exploitations is due to: the easiness of finding and exploiting such vulnerabilities; 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, developers normally focus on functionalities and user requirements, and tend to neglect security aspects due to time constraints.

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. Web application developers and system administrators often rely on them to test web applications against vulnerabilities. Therefore, for them, 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 assessing the performance of these scanners using the proposed methodology.

2 Security Risk and Vulnerabilities

Security risk assessment is fundamental to the security of any organization. It is essential in ensuring that controls and expenditure are fully commensurate with the risks to which the organization is exposed.

The risk deliberated can be defined by the following formula:

*Risk=Likelihood of the threat*vulnerability*consequences of the exploitation*

A threat is, in a general approach, anything that might trigger a risk. However, it is important to point out that a threat is effective only if it is connected to a vulnerability. The risk is thus dependant on the vulnerability and on the threat. Threats are mitigated through vulnerability analysis over the assets. According to the vulnerability analysis, the threats can be eliminated or reduced to a point where the value of the risk is acceptable.

This chapter is therefore intended to explore vulnerability analysis as one of the basic elements of risk, and to introduce vulnerability injection to help ensure compliance with security policies, external standards and with legislation.

2.1 Web application vulnerability scanners benchmarking approach

The approach to evaluate and benchmark the web application vulnerability scanners consists of injecting software faults into a web application code and checking if the scanners can detect the potential vulnerabilities created by the injected faults. The existence of vulnerabilities is confirmed manually in order to get accurate measures of the detection coverage and false positives. The characteristics of the faults injected are derived from the adaptation the web application environment of generic software faults not related with security issues, resulting from a field study [Durães06].

2.2 Web application testing methodology

Web application vulnerability scanners execute their procedures based on the knowledge of a large collection of signatures of known vulnerabilities, 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 XSS and SQL Injection. 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 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 vary a lot. This is actually one of the reasons why it is so important to have means to compare vulnerability scanners.

Two of the most widely spread and dangerous vulnerabilities in web applications are XSS and SQL Injection, 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.

The proposed methodology assumes typical topologies of web application installation and web servers. In a common setup, we need two computers connected by an Ethernet network. One computer acts as a server executing the functions of a web server, an application server and a database server. For the evaluation of server side security mechanisms like web application firewalls, IDSs, it is in this computer where they run. The other computer acts as a client with a web browser. For the evaluation of client side security mechanisms like web application vulnerability scanners, it is in this computer where the scanners are executed.

The methodology of injecting software faults into a web application, one fault at a time, consists of three main stages described in the following paragraphs.

2.3 First Stage

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. This proposal is based on the G-SWFIT software fault injection technique [Durães06] focusing on the emulation of the most frequent types of faults.

Although the G-SWFIT fault operators were also evaluated for other languages, none of them are typical programming languages used for the development of web applications (usually scripting languages, like PHP or PERL). Thus, small adaptations in the fault operators proposed had to be introduced to use them for our web application purposes. The biggest change was in the "Missing function call (MFC)" operator. In web application programming there are normally lots of functions subject of security problems that process a parameter and returns data that will be used by the program. For example, in PHP code it is quite common to have code like this:

```
<? 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 [Durães06] it is not possible to insert this kind of fault because it fails to follow the restriction of the MFC rules. The MFC 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 and created a new operator named "Missing function call extended (MFCext.)".

When the list of faults that can be injected in a web application 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.

Fault type	Description	% of total observed in the field	ODC class
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
MVIV	Missing variable initialization using a value	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 %**	

Table 1 - Most frequent software fault types, derived from a field work (adapted from [Durães06]).

2.4 Second Stage

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 security tools under assessment and their results are gathered.

The testing of a client side security mechanism, like web application vulnerability scanners starts, 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.

Because of the existence of (at least) two computers, some operations need to be performed in the server computer and some in the client computer, in synchronism and this is guaranteed by a Control Tool specially developed for this operation. After the “gold run”, the Control 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 (Figure 1):

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

2. The next fault is injected into the web application.
3. The web application vulnerability scanner is started and at the end, the results are saved into a file. The file name includes a reference to the web application file and the type of fault injected. The Control 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 until all the web application vulnerability scanners have been evaluated.

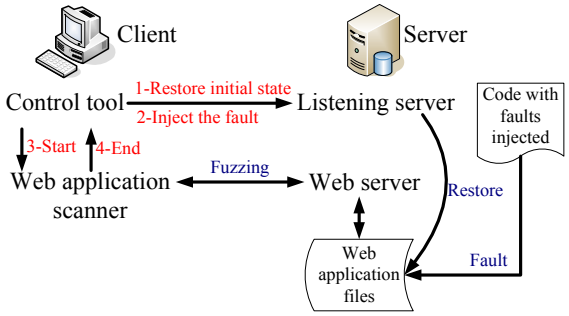

Figure 1– View of the client and server algorithmic procedures.

2.5 Third Stage

Finally in the Third Stage, the resulting data is analyzed in order to obtain a comparative evaluation of the security tools. This procedure can be used, for example, to compare the detection capabilities of web application vulnerability scanners, WAFs, IDSs, etc.

After all tests have been performed, every file resulting from the execution of the scanners is manually analyzed using the algorithm presented in Figure 2. This data convey the decisions of the scanners regarding every vulnerability that was injected. Their results must be analyzed in order to be classified.

In these experiments, we are only interested in XSS and SQL Injection vulnerabilities, so when the scanner reports other types of vulnerabilities they are ignored. All the reported vulnerabilities are manually 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".

To verify the accuracy of the scanners, it is possible to test if they found every vulnerability present in the web application, or to test if they found every trigger of every vulnerability. The former test allows comparing the scanners by the number of alarms raised. However, a scanner can be able to find more places that trigger a given vulnerability and fail to detect other vulnerabilities, while another scanner may find more vulnerabilities, even if it does not detect every input places where these vulnerabilities can be triggered. For practical reasons it was considered this later results, because they are more accurate for the corrections purpose. This is the main

objective of the scanners: to allow the developers to correct the flaws of the web application. For this case, the vulnerabilities are also verified manually to confirm that they are unique and not the same vulnerability tested in a different way. This may happen when the same vulnerable source code is executed even when called from different places in the web application interface. For instance, when we press the "Insert" button or the "Update" button in a HTML `FORM` they may execute some common code. If the vulnerability is in the common code both actions will be triggering the same vulnerability and it should only be accounted only once.

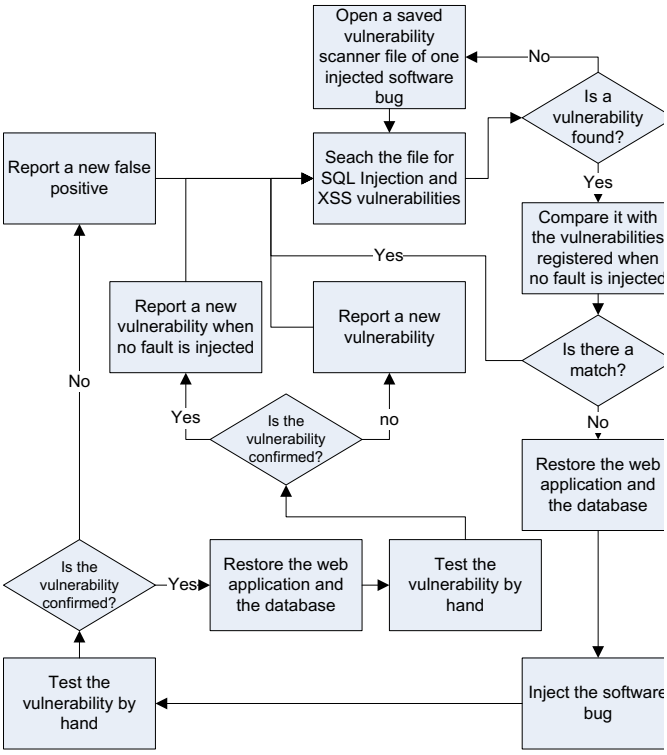

Figure 2 - Algorithm applied to the scanner generated files.

3 Assessing scanners for XSS and SQL Injection

For the evaluation experiments of web application vulnerability scanners 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 were under test: the Acunetix Web Vulnerability Scanner 4, the Watchfire AppScan 7 and the Spi Dynamics WebInspect 6.32.As test bed web application we used a custom-made personal reference information manager called MyReferences. It allows the storage of pdf documents and information about their title, authors and year of publication, for example. The underlined database used consisted in 114 publications from an overall of 311 authors. The web application code has 12 PHP files with 1,436 lines of code.

3.1 Overall results

For the experiments with the MyReferences web application we injected the 12 most frequent types of faults described in Table 1 and derived from the results of a field study on common software bugs [Durães06]. Every source code file was analyzed, looking for possible locations for each fault type. We injected 659 faults and we executed the scanners looking for them. The detailed results of the experiments are depicted in Table 2.

Fault Types	# Faults	Acunetix		AppScan		WebInspect		Total distinct vulnerabilities found by scanners			
		XSS	SQL	XSS	SQL	XSS	SQL	XSS	SQL	#	%
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%
MFCext.	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	**659**	**25**	**33**	**8**	**21**	**19**	**66**	**49**	**83**	**118**	**18%**

injected												

Table 2 – Detailed results.

The faults injected produced application bugs and application malfunctioning, but they also produced a considerable amount of security vulnerabilities, 18%. Note that some injected bugs contributed to more than one type of vulnerabilities (XSS and SQL Injection) and some produced more than one vulnerability of the same type.

One aspect that should be highlighted is the high number of vulnerabilities found even before the start of the tests (they are latent errors). These are the 14 vulnerabilities that were present before any fault was injected by the experiments.

3.2 XSS and SQL Injection comparison

Table 2 shows that, from the 12 fault types only six produced vulnerabilities. These fault types are the “Missing "If (cond) { statement(s) }" (MIFS)”, the “Missing function call extended (MFCext.)”, the “Missing "AND EXPR" in expression used as branch condition (MLAC)”, the “Missing "if (cond)" surrounding statement(s) (MIA)”, the “Wrong logical expression used as branch condition (WLEC)” and the “Wrong variable used in parameter of function call (WPFV)”. Every one of these six fault types generated both XSS and SQL Injection vulnerabilities.

The distribution of XSS and SQL Injection is shown in Table 3. Fault injection produced more than the double of SQL Injection type than XSS.

	XSS	SQL Injection
#	37	81
%	31%	69%

Table 3 - Type of vulnerabilities of the MyReferences application.

3.4 Coverage

The analysis of the individual results of the scanners shows that all the scanners have detected some vulnerabilities that none of the others have. After having the data supporting this conclusion, we suspected that the scanners might leave some vulnerabilities undetected, which is also stated by other studies [Ananta09]. To search for the vulnerabilities left undetected by the scanners and, therefore, analyze the scanners coverage, a human tester was used to perform a manual inspection of both the PHP code and the browser results.

The overall coverage is depicted in Figure 3. The intersection area of the circles represent vulnerabilities detected by more than one scanner. The actual number of vulnerabilities detected is also shown.

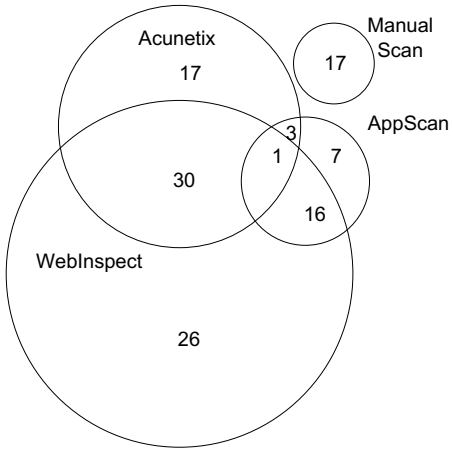

Figure 3 - Total coverage of the MyReferences application.

Analyzing Figure 3 we can see that the circle representing the manual scan does not intersect the other circles, which means that the vulnerabilities detected by manual inspection were not detected by any of the tools evaluated. The radius of each circle is proportional to the number of vulnerabilities detected, providing a comparative visual image of the coverage of each tool. The observation of Figure 3 clearly shows that WebInspect is the best scanner concerning overall coverage of vulnerability detection, followed by Acunetix and AppScan.

The manual scan detected 17 vulnerabilities that have not been detected by none of the vulnerability scanners, which corresponds to 9% of all vulnerabilities found. For the BookStore application, a complete hand scan could not be done due to time constraints, however some quick tests uncovered the existence of some second order vulnerabilities that were not detected by the scanners, which confirms the trend observed in the MyReferences experiments.

Looking at the details of the coverage of the individual vulnerability types (Figure 4 for XSS and Figure 5 for SQL Injection) it is possible to conclude that the best scanner for SQL Injection is not necessarily the best for XSS.

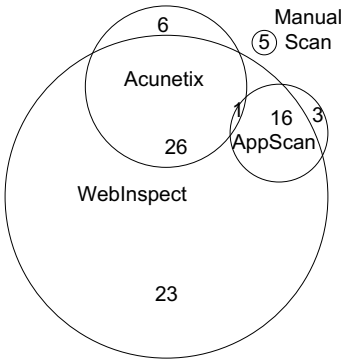

Figure 4 – SQL Injection coverage of the MyReferences application.

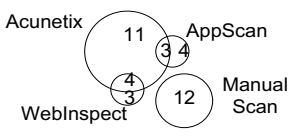

Figure 5 – XSS coverage of the MyReferences application.

Given the high price of these commercial scanners, they leave many vulnerabilities undetected. While some of these vulnerabilities should have been detected by the scanners, there are others that will be difficult to be detected by a tool using only the black-box approach. Other type of vulnerabilities undetected are logic errors and second order vulnerabilities, which are vulnerabilities that need some reasoning to detect them. Although a human tester can uncover them, they are not easily automated (and implemented by the scanners) and generalized for every web application.

Another difficulty for the scanners occurs 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.

3.5 False positives

The scanners found some vulnerabilities but they also detected many false positives, as depicted in Table 4. Like in many other related fields, the false positive rate tends to be directly proportional to the ability to detect vulnerabilities.

	Acunetix	AppScan	WebInspect
#	13	43	45
%	20%	62%	38%

Table 4 - False positives of the MyReferences application.

We also analyzed the possible reasons for the false positives to provide some insights on how the scanners could be improved. Some false positives occurred due to an error issued by the web application in normal execution because of the fault injected. In the penetration test, the same error was shown and that triggered the scanner. This error message was found in 10 cases using the Acunetix, in 43 cases using the WebInspect, and in 40 cases using the AppScan. We could not reproduce the other three remaining cases of false positives found by Acunetix and the two remaining by WebInspect. The three remaining false positives found by AppScan were curiously triggered by the data stored in the back-end database: the cause was the title of a paper about SQL Injection.

4 Conclusion

In this chapter we proposed an approach to evaluate and compare web application vulnerability scanners, in order to eliminate the threats or reduce them to a point where the value of the risk is acceptable. 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. Even the common widely used Rapid Application Development environments produce code with vulnerabilities. For some critical web applications several scanners should be used and a manual scan should not be discarded from the process. In fact, it should be mandatory for critical applications.

Each one of the web application vulnerability scanners analyzed cannot be used as a "One tool to rule them all" solution. Even the results of the three scanners combined do not cover the vulnerabilities thoroughly. Through a different set of experiments, using PHP, Java, ASP.NET and ASP applications and also testing for JavaScript related problems, Ananta Security compared the same brand scanners and their conclusions are similar to ours [Ananta09]: the scanners have a huge false positive rate and the black-box scanning using automated tools is not enough to assure complete security. The disturbing conclusion is that, even if the scanners do not find any vulnerability we cannot assure that the web application is free of vulnerabilities.

References

[Acunetix12] Acunetix, Acunetix Web Security Survey Report, Acunetix, (2007). available from: http://www.acunetix.com/news/security-audit-results.htm

[Ananta09] Ananta Security, Web Vulnerability Scanners Comparison, (2009), available from: http://anantasec.blogspot.com/2009/01/web-vulnerability-scanners-comparison.html

[CodeCharge07] CodeCharge, Online Bookstore Web Appplication, available from: http://www.gotocode.com/apps.asp?app_id=3

[Durães06] Durães, J., and H. Madeira, Emulation of Software Faults: A Field Data Study and a Practical Approach, IEEE Transactions on Software Engineering, 32(11), 849-867, doi:10.1109/TSE.2006.113, (2006)

[Gordon06] Gordon, L. A., M. P. Loeb, W. Lucyshyn, and R. Richardson, 2006 CSI Computer Crime & Security Survey, Computer Security Institute, (2006)

[McGraw08] McGraw, G., Software [In]security: Software Security Demand Rising, (2008), InformIT. available from: http://www.informit.com/articles/article.aspx?p=1237978

[MITRE12] MITRE Corporation, Common Vulnerabilities and Exposures, 2012, available from: http://cve.mitre.org/

[OWASP10] OWASP Foundation, OWASP Top 10 – 2010, OWASP Foundation, (2010)

[Winkler10] Ira Winkler, Justifying IT Security Managing Risk & Keeping Your Network Secure, Qualys Inc.

[YesSoftware09] YesSoftware, "CodeCharge Studio 4.2", (2009), available from: http://www.yessoftware.com/products/product_detail.php?product_id=1