Training Security Assurance Teams using Vulnerability Injection

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

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

Abstract

Writing secure web applications is a complex task. In fact, a vast majority of web applications are likely to have security vulnerabilities that can be exploited using simple tools like a common web browser. This represents a great danger as the attacks may have disastrous consequences to organizations, harming their assets and reputation. To mitigate these vulnerabilities, security code inspections and penetration tests must be conducted by well-trained teams during the development of the application. However, effective code inspections and testing takes time and cost a lot of money, even before any business revenue. Furthermore, software quality assurance teams typically lack the knowledge required to effectively detect security problems. In this paper we propose an approach to quickly and effectively train security assurance teams in the context of web application development. The approach combines a novel Vulnerability Injection Technique with relevant guidance information about the most common security vulnerabilities to provide a realistic training scenario. Our experimental results show that a short training period is sufficient to clearly improve the ability of security assurance teams to detect vulnerabilities during both code inspections and penetration tests.

1. Introduction

A large number of web application developers do not have the required software engineering skills and competences to build secure code. The consequences can be disastrous as there is a wide collection of vulnerabilities affecting many web sites that can be maliciously explored. Aggressive time to market constraints and reduced cost policies push companies to release their software as soon as possible, disregarding, in many cases, the quality assurance procedures needed to identify and mitigate potential code vulnerabilities.

To reduce the number of security vulnerabilities, web applications must undergo quality assurance procedures, including Code Inspections (the application is analyzed from the point of view the programmer) and Penetration Testing (the application is tested from the point of view of the users/attackers) [1]. Although these procedures are mandatory for companies that want to be compliant with security standards like the Payment Card Industry [2], that is not the case for the vast majority of web applications in the field. Furthermore, as quality assurance teams frequently lack the knowledge required to detect security problems, they tend to focus only on typical software bugs, leaving many security vulnerabilities undetected, which may not only have a devastating cost to the company but also to their clients.

In this paper we propose a methodology to train security assurance teams to perform effective code inspection and manual penetration testing in web applications. Our approach uses the injection of realistic vulnerabilities in web application files that are going to be used during training activities. This provides to security teams an experience close to what they will find when inspecting or testing web applications to detect real vulnerabilities. The vulnerabilities injected are realistic as they are defined based on the results of a field study on real security vulnerabilities [3]. With this data we built a Vulnerability Injection Tool that automates the injection process.

In our proposed methodology, a security assurance team should start by attending a short generic training course on security in web applications followed by a practical exercise in which the team tries to find vulnerabilities in software code. Afterwards, the team attends another short training course, this time focusing on providing the team relevant information on the most common vulnerabilities found in web applications. As a final step the team performs a second practical exercise on security code inspection and penetration testing (obviously, the team is expected to perform better during this exercise as a result of the knowledge they acquired during the second training course). The code used during the practical exercises is generated by automatically injecting vulnerabilities in the source files of web applications using the Vulnerability Injection Tool.

We have tested our approach in laboratory to assess its effectiveness. Two different teams attended the training sessions and results show that both teams increased their ability to detect vulnerabilities. To have a more detailed view on the performance of the teams, we also executed some penetration tests using commercial vulnerability application scanners. Amazingly, both security teams outperformed the vulnerability scanners by detecting more vulnerabilities.

The structure of the paper is as follows. Section 2 presents the proposed approach. Section 3 explains how the vulnerability information is used by the Vulnerability Injection Tool and during the training sessions. Section 4 presents the experiments performed and discusses the results. Section 5 concludes the paper and suggests future work.

2. Methodology for Training Security Assurance Teams

Security concerns must be present during all the phases of the software development lifecycle and security cannot be seen just as a minor issue. In fact, it must be a design goal [4] and this is represented well in Microsoft's [5] and McGraw's [6] software development lifecycles. Security vulnerabilities must be mitigated during the development lifecycle before the software is released. **Code Inspection** and **Penetration Testing** represent two key quality assurance procedures that must be used to detect security vulnerabilities. Code inspection is a white-box approach that consists in the formal review of the application code by an external team. Penetration testing is a black-box approach consisting in a set of tests made from the point of view of the users, where the external team tries to find all the possible vulnerable entry points of the application. Penetration testing can be performed manually or it can be done using automated tools, although even top commercial products have a high rate of false positive (non vulnerabilities that are tagged as vulnerabilities) and false negative (vulnerabilities that are not identified) values [7].

Searching for security vulnerabilities is different from searching for generic software bugs. When searching for bugs the objective is to see if the code is compliant with the functional specification of the application. It is common to forget to analyze the consequences of unspecified situations and usually this leads to undetected security problems. Searching for security vulnerabilities, on the other hand, is aimed at probing for dangerous hidden functionalities that are somehow present in the code and that can be maliciously exploited.

The methodology proposed in this paper is aimed at the training of the security assurance teams for code inspection and penetration testing and consists of four key steps:

1. Basic training. The team attends a short generic training course about vulnerabilities in web applications and how to detect them using both source code inspection and penetration testing. In this session no detailed information is given about the profile of typical security vulnerabilities.

2. First test. The second step is a practical training session to consolidate what was learned and to get a baseline measure of the performance of the team (concerning vulnerabilities identification) before being specifically trained for security vulnerabilities identification (the next step). To create a lifelike scenario, a set of realistic vulnerabilities are injected in the web application files that are going to be used. These vulnerabilities are based on the most common vulnerabilities found in web applications and the injection can be done using the Vulnerability Injection Tool proposed in the present paper (see 3.3 for details).

3. Specific training. The team attends another short training course. This course focuses on the specific attributes of the most common vulnerabilities found in web applications, like where they may be located and what code is usually responsible for them (see 3.1 for details). It also provides guidance on how to exploit these vulnerabilities based on the specific characteristics of the vulnerabilities (see 3.2 for details).

4. Second test. At the end there is a second practical training session to consolidate all that was learned. The setup is similar to the one used in the first practical training session. The number of vulnerabilities detected by the security team and the time needed to detect them are important metrics that are used to evaluate if the team’s ability to identify security vulnerabilities improved when compared to the first practical session. These metrics are collected and analyzed separately for each type of quality assurance procedure (code inspection and penetration testing).

Searching for vulnerabilities of every type in web application code is time consuming and requires high expertise on the huge variety of code patterns that represent vulnerabilities. It is much simpler and cheaper to search only for the most common vulnerability types, if they can cover almost all the situations. If we can quickly and easily mitigate this type of vulnerabilities, we are cleaning the most important security problem in web applications.

3. Vulnerability Data and Vulnerability Injection Tool

In order to focus on the most common types of vulnerabilities affecting web applications we use the results from a field study that classified 655 security patches of six widely used LAMP (Linux, Apache, MySQL and PHP) web applications [3]. This field study focuses on **Cross Site Scripting** (XSS) and **SQL Injection** vulnerabilities. Note that these are two key vulnerabilities that, together, were responsible for approximately 1/3 of all the Common Vulnerabilities and Exposures in 2006 [8].

The fault types that resulted from the field study [3] are depicted in Table 1, along with their distribution. As we can see, the MFC extended fault type is the most common, accounting for most of the vulnerabilities found. It represents vulnerabilities that are caused by a variable that was not properly sanitized and that should have been cleaned by a specific function (which the programmer did not include in the code).

This way, to build a realistic vulnerability injector for web applications we do not need the specific characteristics of each one of the 12 fault types of Table 1. In fact, because the MFC extended fault type is responsible for 76% of all the security problems analyzed and the next fault type is as low as 7%, the MFC extended is the obvious candidate to be used in a first approach to build a tool for vulnerability injection.

The MFC extended is typically observed in situations where the missing function is related to filtering or changing the content of one of its arguments. The target argument is a variable whose value comes from a GET or POST HTTP parameters or from database results.

The nature of the function that the programmer failed to include in the source code causing the MFC extended vulnerability is determinant for this fault type analysis. This is why in [3] this fault type was divided into the sub-types **A**, **B** and **C**, accounting for 45.34%, 18.32% and 12.21% of all the vulnerabilities, respectively:

A - Missing casting to numeric of one variable. The missing function converts a PHP variable to numeric type. This can be accomplished with the “(int)” type cast or the “intval” PHP function. Although the “(int)” type cast is not really a function, it is present in this sub-type because it behaves just like the “intval” function.

B - Missing assignment of one variable to a custom made function. To cope with specific needs of cleaning PHP variables from code injection, the software programmer may have to write its own filtering functions. This fault type refers to the situations where the programmer forgets to use one of these specific functions with some critical variable.

C - Missing assignment of one variable to a PHP predefined function. The missing function is one of the PHP predefined functions that can be used to filter variables from code injection. According to the field study presented in [3], the most frequent PHP predefined functions related to this vulnerability type are the following: “addslashes”, “eregi_replace”, “stripslashes”, “htmlentities”, “preg_replace”, “htmlspecialchars”, “md5”, “str_replace”, “urlencode”.

3.1. The MFC extended Vulnerability Operators

The Vulnerability Injection Tool uses information about the **Location Pattern** and **Vulnerability Code Change** of each vulnerability type in order to inject vulnerabilities in the source code of web applications. These attributes are the building data for the Vulnerability Operator and they define how to inject the fault type in the web application source code. They

Table 1. The vulnerability fault types observed in the field and their relevance [3]

Fault type	Description	Vulnerabilities (%)
MFC extended	Missing function call extended	75.88
WPFV	Wrong variable used in parameter of function call	7.02
MIFS	Missing if construct plus statements	5.19
WVAV	Wrong value assigned to variable	4.27
WFCS	Wrong function called with same parameters	2.75
MVIV	Missing variable initialization using a value	1.37
MLAC	Missing "AND EXPR" in expression used as branch condition	1.37
EFC	Extraneous function call	0.92
MFC	Missing function call	0.61
MIA	Missing if construct around statements	0.31
MLOC	Missing "OR EXPR" in expression used as branch condition	0.15
ELOC	Extraneous "OR EXPR" in expression used as branch condition	0.15

are also used during the second training course provided to the teams.

In spite of the valuable information provided by the field study presented in [3], the results are not enough to completely define the Vulnerability Operators. We need additional data targeting on the location patterns and on the code fixes of the vulnerabilities. To obtain these attributes we conducted another field study in which we analyzed the same raw data of the 655 code fixes from the 6 web applications used by [3], focusing on how to mimic the vulnerabilities found in the code. Due to space restrictions, only the information related to the MFC extended fault type is described in this paper:

Location Pattern:

To inject MFC Extended vulnerabilities we need to locate functions having the following characteristics:

- The function must be one of the functions defined in the sub-types A, B or C (the functions targeted depend on the sub-type being injected);

- The argument of the function is directly or indirectly related to an input value from the outside: POST, GET, the return of an SQL query;

- The output of the function is going to be displayed on the screen or is going to be used in a POST, a GET variable or is going to be used in a SQL query string;

- The function can be an argument of another function or have another function as the argument;

- In the argument of the function, the vulnerable variable may also be present inside a $_GET, $HTTP_GET_VARS, $_POST, $HTTP_POST_VARS PHP variable arrays;

- For the MFC extended sub-types B and C the vulnerable variable may be one of the PHP variables, like the $_SERVER['PHP_SELF'].

Vulnerability Code Change:

After finding the potential locations for the vulnerability injection we need to perform the code change. Depending on the code surrounding the function, one (and only one) of the following changes is applied in the case of MFC Extended vulnerabilities:

1. If the function is used in an assignment as the only line of code and the variable is not inside a $_GET, $HTTP_GET_VARS, $_POST, $HTTP_POST_VARS PHP variable array the whole line of code is removed. For example remove the line "`$vuln_var = intval($vuln_var);`";

2. If the function is used in an assignment as the only line of code and the variable is inside a $_GET, $HTTP_GET_VARS, $_POST, $HTTP_POST_VARS PHP variable array only the function is removed from the code, leaving the argument intact. For example to replace "`$vuln_var = intval($_GET['vuln_var']);`" with "`$vuln_var = $_GET['vuln_var'];`";

3. In the other cases only the function is removed leaving in the code only the variable, or the $_GET, $HTTP_GET_VARS, $_POST, $HTTP_POST_VARS PHP variable array if the variable is inside. For example to replace "…`"'str1'.intval($vuln_var).'str2'";`" with "…`"'str1'.$vuln_var.'str2'";`".

An important aspect is that the injection of these code changes does not prevent the application from running. In fact, the web application code continues to run without any syntactic or execution errors (except for the vulnerability injected).

3.2. Code exploit

The Vulnerability Operators also provide some insights that can help a code inspection or penetration test team to find and exploit the vulnerabilities emulated. Looking at the MFC extended Vulnerability Operator we can see that integer variables are a major target. They are derived from the sub-type A and responsible for 45.34% of all the vulnerabilities found. During code inspection the security team should verify thoroughly if all the variables that should store integer values cannot contain a non integer value (assigned, displayed, etc.). During penetration testing these variables with integer values should be a primary target to attack. This can be accomplished by appending a text to them, for example:

- "` or 1=1`" or "` or 'a'='a'`", etc. when searching for SQL Injection;

- "`<script>alert('XSS')</script>`" when searching for XSS;

- Fuzzing with `'`, `"`, `>`, `<`, `)`, `(`, etc. at the begin and/or end of the variable text.

Sub-types B and C have more functions to target than sub-type A, so the information that can be derived from them is more generic. During code inspection the security team should search for variables that are not sanitized using PHP functions (sub-type C) or functions developed specifically for the application (sub-type B). The possible use of the non-sanitized PHP variables, like the "`$_SERVER['PHP_SELF']`" should also be tested. In this case, the test is typically done by attaching a XSS exploit at the end of the script name and path in the URL. For example, the link: "`http://test.com/index.php`" could be tested with:
"`http://test.com/index.php/"><script>alert('XSS')</script>`".

It is important to emphasize that this code exploit type of information is presented to teams during the second training session in our methodology. An explanation of the typical approaches to fuzz for SQL Injection [9] and XSS [10, 11] is done in the first training session.

3.3. The Vulnerability Injector Tool

We implemented a **Vulnerability Injector Tool** based on the Location Pattern and Vulnerability Code Change attributes of the Vulnerability Operator. Currently it only supports the three MFC Extended sub-types, but others can be easily configured. To use this tool we must start by specifying the path to a PHP file of a web application. After, the procedure for injecting software faults consists of two stages, which are executed automatically by the tool:

1. 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 fault locations and vulnerability types. When the list of potential locations is extensive, because the application code is large, resulting in lots of possible locations for each fault type, the relative percentages shown in Table 1 are used to select locations.
2. Each fault is injected, which corresponds to the insertion of the Vulnerability Code Change (defined by the Vulnerability Operator) in the web application file. The result of this process is a set of delta files containing the vulnerability locations found. The delta files consist of only the modified portion of the PHP code with its location, making it easier for a person to analyze and store it. The delta files may be applied to the original file (injecting the vulnerabilities) by using the Unix "patch" command, that is also available for other Operating Systems.

4. Experiments with Code Inspection and Penetration Testing

In this section we describe the training experiments and discuss the results. The experiments focus on both **Code Inspection** and **Penetration testing** and the key objective is to verify if the training based on the knowledge of the most common vulnerabilities improves the detection skills of the security assurance teams. The other objective is to confirm the usefulness of the Vulnerability Injection Tool in providing web application files with vulnerabilities suitable for training the teams.

The **Code Inspection** test consists of the execution of a formal code inspection procedure targeting a block of source code of a web application. This block of code was previously injected with vulnerabilities using the Vulnerability Injector Tool. In this formal code inspection procedure each member of the team has its own role, as in traditional code inspections [12]: a Moderator, a Reader, a Note Taker and the others are Inspectors. The Author of the code is also present to clarify any doubts about the web application.

The **Penetration testing** consists of interacting with the web page of the application from the point of view of the attacker. The test team searches for vulnerabilities by trying to penetrate the application tweaking the POST and GET HTTP parameters, although they do not know the source code being executed. The web page under attack was previously injected with vulnerabilities using the Vulnerability Injector Tool. During the Penetration Testing the data in the database may change as a result of the natural fuzzing process to find vulnerabilities. This is usually the case when searching for SQL Injection vulnerabilities, because the tester is tweaking the SQL queries sent to the database. To restore the database to its initial state we built the **Vulnerability Injector Remote Control Application**. This application communicates with a service deployed in the database server computer. This service is listening in a specific port and is able to restore the database when requested by the control application.

Both code inspection and penetration testing teams follow the experimental procedure presented in Section 2. The following points present some information on the instantiation of the procedure for these concrete experiments:

1. Basic training. It is a thirty minute generic training on XSS and SQL Injection. This training is based on data from the Open Web Application Security Project (OWASP) [13]. In this training session we describe the vulnerabilities, what causes them (the deficient validation of external input) and the dangers involved. Then, we explain the generic ways to search for them using the source code of the web application and using the browser by looking for the display and for the HTML generated. One real life example of exploiting each type of vulnerabilities is also detailed.

2. First test. The security team executes the first set of code inspection and penetration tests.

3. Specific training. It is also a thirty minute training on XSS and SQL Injection, but this time focusing on the results of the field study [3], on the Vulnerability Operators (Location Pattern and Vulnerability Code Change) and Code Exploit that are described in sections 3.1 and 3.2.

4. Second test. The security team executes a second set of code inspection and penetration tests. These tests target a block of code different from the one used in step 2.

For the experiments we used the MyReferences custom made web application. It consists of 13 PHP files and it runs in a Linux Box with the Apache Web

Server accessing a MySQL database. This application is used to manage publications: it allows the storage of PDF documents, and some information about them like the title, the conference, the year of publication, the document type, the relevance, and the authors. The database used comprises five tables and has currently stored 118 publications from 317 authors.

For the experiments we used two security teams of six elements each. One of the teams (team T1) incorporated experienced people with several years of software development, including a technical manager, a quality assurance officer, and a project manager. The other team (team T2) was composed of computer science university students without much programming experience. In what concerns the vulnerabilities tested, some of them have some incipient knowledge about SQL Injection but they all have very little or none about XSS.

The people involved in our experiments can be considered as non security experts, as none of them had ever been part of a security test team, although they have some insights of the technologies involved. As the main goal of the experiments is to evaluate the learning curve provided by our approach of training people using vulnerability injection scenarios, the low level of expertise on security coding was not a problem. Unfortunately, the reality is that many web application projects actually use programmers without specific know how on secure coding, just like the two teams used in our experiments. In this sense, the results of the experiments also represent what can be achieved in training mainstream web programmers.

Four days before the start of the experiments we provided the teams a document detailing the web application files and the entity-relationship diagram of the database. Furthermore they had access via a web browser to the web application and they knew the login credentials of a registered user of the web application.

4.1. Code Inspection Experiments

For the code inspection tests we used the following two files of the web application:

- **edit_paper.php**. Allows the update, delete insert and visualization of the information of to each paper;
- **show_papers.php**. Shows the information about a list of papers that can be sorted by any field. Each page only shows five papers at a time and it is possible to filter the papers using some restrictions.

We picked two different blocks of code from the edit_paper.php and injected 4 vulnerabilities in each. The number of vulnerabilities injected respected some rules: must be the same number in each block; it must allow the team to be able to get results even if they missed some vulnerabilities; it must be a reasonable number for the total of lines of code considered (in this case approximately 100 lines of code). The same procedure was done to the show_papers.php but we were able to inject 5 vulnerabilities in each one. In order to expose similar code in both periods, one block from each file was used during the Basic Training period and the others during the Specific Training period.

The results of the Basic Training Period of code inspection done by the two teams (T1 and T2) are depicted in Table 2. We can observe the number of vulnerabilities injected in the files, the number of vulnerabilities discovered and the average time spent analyzing each line of code. The results of the Specific Training Period are depicted in Table 3.

Comparing the results obtained before and after the Specific Training we can see a clear improvement in the number of vulnerabilities discovered by the code inspection security teams. In the first training period both teams discovered 5 vulnerabilities and left 4 undetected. After the Specific Training, they could find all the 9 vulnerabilities injected. An interesting aspect is that both teams were able to find more vulnerable locations than those that were expected. These are represented with a (+nº) in Table 3. This enforces the idea that we can never know when we have all the vulnerabilities mitigated, although it is important to address the most that we can, thus reducing the attack surface. An important aspect is that, although the security teams were much more effective in the second training period, they spent nearly the same amount of time inspecting each line of code than before.

Table 2 – Code Inspection results of the basic training period

Web Application File	Code Lines	# vulnerabilities			#Seconds/Line of code	
		Injected	Discovered			
			T1	T2	T1	T2
edit_paper.php	1-104	4	3	2	18	51
show_papers.php	36-184	5	2	3	16	30
	Total	**9**	**5**	**5**	**17**	**33**

Table 3 – Code Inspection results of the specific training period

Web Application File	Code Lines	# vulnerabilities			#Seconds/Line of code	
		Injected	Discovered			
			T1	T2	T1	T2
edit_paper.php	105-215	4	4	4	23	24
show_papers.php	185-283	5	5 (+4)	5 (+1)	13	28
	Total	**9**	**9** (+4)	**9** (+1)	**18**	**25**

Note: Unexpected vulnerabilities that were discovered are represented by a (+nº)

The teams also made some mistakes in these experiments. During the Basic Training period team T1 reported a variable as being vulnerable in the show_papers.php. Although this variable is not sanitized in the code, all the possible values that it may have belong to a set of hard coded values, making it impossible to be exploited by an attacker. During the Specific Training period team T1 also detected the use of the same variable responsible for the previous mistake in the same PHP file in three other locations. As expected, they signaled these as possible locations to be exploited. This mistake was clearly propagated from the previous code inspection phase. Both teams indicated another variable as being vulnerable to attack (in the edit_paper.php file), but again that variable could only take values hard wired in the code. It is a good practice to sanitize every input variable, and all mistakes that were found in the two phases are fine recommendations for the programmers. Although they are not currently a threat, a future upgrade of the web application can change some parts of the source code exposing these unprotected variables to the attacker.

4.2. Penetration Test Experiments

For the Penetration Test Experiments we have used as target the file **edit_authors.php**. This file is responsible for the update, delete insert and visualization of the information related to the authors of each paper. We created two modified versions of this file, one to be used during the Basic Training period and another to be used during the Specific Training period. In each of the modified versions we injected 5 vulnerabilities guaranteeing that those injected in one version were different than those injected in the other version. Once again, the vulnerabilities used were provided by the Vulnerability Injection Tool.

We restricted the search time to 60 minutes for each training period of penetration testing, because this time should be enough for the teams to find most of the vulnerabilities injected without dwindling the team's detection efficiency. The chosen target application file used only GET HTTP parameters, preventing the unnecessary need for more time to perform the tests, as if it had POST HTTP parameters the teams had to use something like a proxy (e.g. Paros Proxy [14]) to be able to intercept the communication between the browser and the Internet. After having intercepted the communication, it is as easy to manipulate POST as GET parameters.

We also wanted to know if the vulnerabilities injected in the edit_authors.php could be detected by some top commercial web application vulnerability scanners and compare the results with those of the security teams. These scanners provide 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. We used the HP WebInspect 7.7 [15] and the IBM Watchfire AppScan 7.0 [16] scanners and we randomly named them S1 and S2. We have decided to keep the brand of the web vulnerability scanners anonymous to assure neutrality and because commercial licenses do not allow in general the publication of tool evaluation results.

The results of the penetration test experiments done to the modified versions of the edit_authors.php file are depicted in Table 4. It includes the data obtained by the two teams (T1 and T2) from both before and after the Specific Training Period. There are also represented the results of the scanners (S1 and S2) when they searched for vulnerabilities in the same web pages used by the teams.

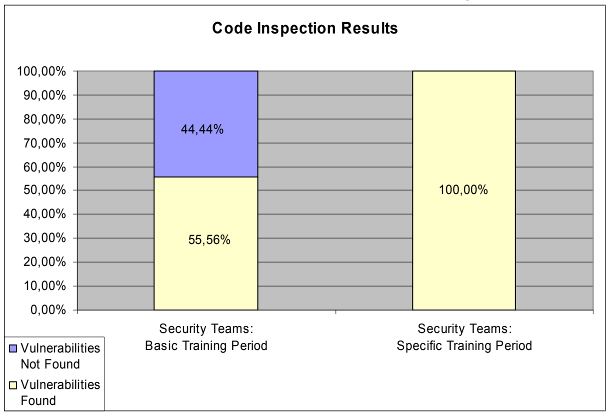
4.3. Overall Results and Discussion

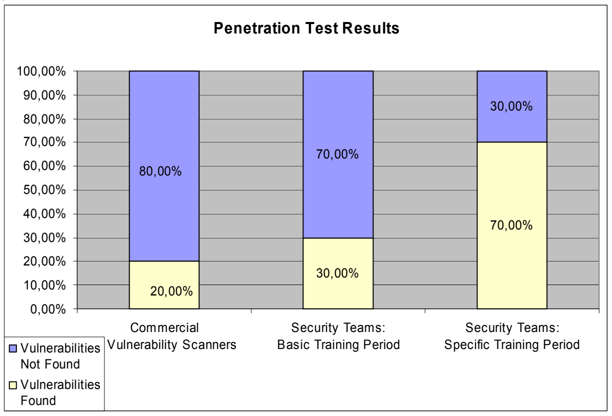
Summing up the results of the Code Inspection Experiments and the Penetration Testing Experiments we observed a clear improvement after the Specific Training (Figure 1). Although the small number of samples we can see that there was an increase in

Table 4 – Penetration Test results

	# vulnerabilities				
	Injected	Discovered and Exploited			
		T1	T2	S1	S2
Basic Training Period	5	1	2	1	0
Specific Training Period	5	4	3	1	2

Figure 1 – Vulnerability Detection Comparison





vulnerability detection of around 40% in both the code inspection and the penetration tests. We can also confirm that the security teams did better than the commercial scanners even before the Specific Training period. These improvements in vulnerability detection are impressive given the short period of time used to train the teams.

5. Conclusion

In this paper we present a methodology for the training of security assurance teams using vulnerability injection. This methodology is based on the most common vulnerabilities found in field study of web applications allowing us to build a Vulnerability Injection Tool (to inject realistic vulnerabilities during the training) and to provide important guidance information to the trainees. The experimental results show an improvement in the number of vulnerabilities found by security teams whether executing code inspection or manual penetration testing. The teams even outperformed top commercial vulnerability scanners in the penetration testing approach. The results emphasize the two main ideas of our approach:

1. The data associated to the most common vulnerability types can be used with success as a guide to train security teams, improving the results of both the code inspection and penetration security tests;
2. The importance of a flexible tool like the Vulnerability Injection Tool to provide files with realistic vulnerabilities to train the security teams.

For future work we intend to use this approach to evaluate security teams, to estimate the total number of vulnerabilities still present in the code and to build a realistic attack injector.

6. References

[1] SAM, NG., "Advanced Topics on SQL Injection Protection". OWASP, 2006
[2] Payment Card Industry (PCI) Data Security Standard, version 1.1, September, 2006
[3] Fonseca, J., Vieira, M., "Mapping Software Faults with Web Security Vulnerabilities", The 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, June 2008
[4] Jayaram, K., Aditya P., "Software Engineering for Secure Software - State of the Art: A Survey", Department of Computer Science, Purdue University, USA, 2005
[5] Howard, M., Leblanc, D., "Writing Secure Code", Microsoft Press, Redmond, WA, USA, 2002
[6] Chess B., McGraw, G., "Software security". IEEE Security and Privacy Magazine, 2004
[7] Fonseca, J., Vieira, M., Madeira, H., "Testing and comparing web vulnerability scanning tools for SQL injection and XSS attacks", PRDC, December 2007.
[8] Steve, C., Martin, R., "Vulnerability Type Distributions in CVE", Mitre report, May, 2007
[9] OWASP, Testing for SQL Injection, http://www.owasp.org/index.php/Testing_for_SQL_Injection H, May, 2008
[10] RSnake, XSS Cheat Sheet, http://ha.ckers.org/xss.html, May, 2008
[11] OWASP, Testing for Cross site scripting, http://www.owasp.org/index.php/Testing_for_Cross_site_scripting, May, 2008
[12] Fagan, M.E., "Design and Code inspections to reduce errors in program development", IBM Systems Journal, Vol. 15, No 3, Page 182-211, 1976
[13] OWASP Foundation, http://www.owasp.org, May, 2007
[14] Chinotec Technologies Company, Paros Proxy, http://www.parosproxy.org/index.shtml, May, 2008
[15] SPI Dynamics Inc., http://www.spydynamics.com/products/webinspect/, May, 2008
[16] Watchfire Corporation, http://www.watchfire.com/, May, 2008