Vulnerability & Attack Injection for Web Applications

José Fonseca[1], Marco Vieira[2], Henrique Madeira[2]
[1]CISUC, University of Coimbra, Polytechnic Institute of Guarda, Portugal
[2]CISUC, University of Coimbra, Portugal
josefonseca@ipg.pt, mvieira@dei.uc.pt, henrique@dei.uc.pt

Abstract

In this paper we propose a methodology to inject realistic attacks in web applications. The methodology is based on the idea that by injecting realistic vulnerabilities in a web application and attacking them automatically we can assess existing security mechanisms. To provide true to life results, this methodology relies on field studies of a large number of vulnerabilities in web applications. The paper also describes a set of tools implementing the proposed methodology. They allow the automation of the entire process, including gathering results and analysis. We used these tools to conduct a set of experiments to demonstrate the feasibility and effectiveness of the proposed methodology. The experiments include the evaluation of coverage and false positives of an Intrusion Detection System for SQL Injection and the assessment of the effectiveness of two Web Application Vulnerability Scanners. Results show that the injection of vulnerabilities and attacks is an effective way to evaluate security mechanisms and tools.

1. Introduction

There is an increasing dependency on web applications nowadays, ranging from individuals to large organizations. Almost everything is stored, available or traded on the web. Web applications can be personal web sites, blogs, news, social networks, webmails, bank agencies, forums, e-commerce applications, etc. The omnipresence of web applications in our way of life and in our economy is so important that they are a natural target for malicious minds.

Security motivation of web applications should reflect the magnitude and relevance of the assets they are supposed to protect. Although there is an increasing concern about security (often being subject to regulations from governments and corporations), there are some significant factors that make securing web applications a task hard to fulfill:

- The web application market is growing fast, resulting in a huge proliferation of web applications, largely fueled by the (apparent) simplicity one can develop and maintain such applications;
- Web applications are highly exposed to attacks;
- It is common to find developers and administrators of web applications without the required knowledge or experience in the area of security.

Not surprisingly, the overall situation of security in web applications is quite favorable to attacks [1, 2, 3]. In fact, estimations point to a very large number of web applications with security vulnerabilities [4, 5] and consequently, there are numerous reports of successful security breaches and exploitations [6, 7]. To fight this scenario we need means to evaluate the security of web applications and attack counter measure tools.

In this paper we address the security of web applications by applying a procedure inspired on the fault injection technique that has been used for decades in the dependability area. In our case, the "security vulnerability" + "attack" represents the space of the "faults" we inject in a web application; and the "intrusion" is the "error" [8, 9]. To emulate with accuracy real world web vulnerabilities we rely on results obtained from a field study on real security vulnerabilities [10] and use them in a novel Vulnerability Injection tool. This tool is, in fact, a key instrument that can be used in several relevant scenarios:

- **Building block of a realistic Attack Injector.** An Attack Injector can be a valuable tool to test various counter measure mechanisms, such as Intrusion Detection Systems (IDS), Firewalls, Web Application Vulnerability Scanners, etc. Conceptually, an attack injection tool consists of the injection of realistic vulnerabilities that are automatically attacked, and finally the result of the attack is evaluated. To verify the success of the attack we need to analyze the flux of information inside the application by placing probes strategically, in the least intrusive way. The analysis of the results of these probes and their synchronism with the attack are key elements of the Attack Injector;
- **Train security teams.** By injecting representative security vulnerabilities in web application code, we provide a realistic test bed for the training of security teams that are going to perform code inspection and penetration testing [11];

978-1-4244-4421-2/09/$25.00 ©2009 IEEE

- **Evaluate security teams.** In a controlled environment, security teams can be assessed based on the number of vulnerabilities they are able to find, the number of false positives reported and the time needed to perform a set of code inspections and penetration tests;
- **Estimate the total number of vulnerabilities still present in the code.** The injection of realistic vulnerabilities in web code can help decide if the software is ready to be released or not. The process consists of injecting vulnerabilities and having a security team searching for them. The team will most likely find some of the injected vulnerabilities and some that already existed in the code. The estimated number of vulnerabilities still present in the software can be obtained from the percentage of those injected that were found and those not injected that were also found, using an approach similar to the one proposed by Steve McConnell for software bugs in general [12].

In this paper, we present a **Vulnerability Injection tool** and an **Attack Injection tool** for web applications, which implement our vulnerability and attack injection methodology. We have implemented these tools and tested them with widely used applications in two case study scenarios. The first goal of our experiments is to evaluate the effectiveness of the Vulnerability Injection tool in producing a large number of realistic vulnerabilities. The second goal is to show how the Attack Injection tool can exploit injected vulnerabilities to launch attacks, allowing the evaluation of the effectiveness of counter measure mechanisms installed in the target system.

The structure of the paper is as follows. Next section presents related research. Section 3 introduces the Attack Injection architecture. Section 4 describes the experiments and discusses the results. Section 5 concludes the paper.

2. Related work

Fault injection techniques have been largely used to evaluate fault tolerant systems [13, 14]. The artificial injection of a large quantity of faults in a system or in a component of the system speeds up the occurrence of faults and errors, allowing researchers and engineers to evaluate the impact of faults on the system and/or potential error propagation to other systems. Fault injection also helps in estimating fault tolerant system measures, such as the fault coverage and error latency [13].

Fault injection techniques have traditionally been used to inject physical (i.e., hardware) faults [13, 14]. In fact, initial fault injection techniques used hardware-based approaches such as pin-level injection or heavy-ion radiation. The increased complexity of systems has lead to the replacement of hardware-based techniques by software implemented fault injection (SWIFI), in which hardware faults are emulated by software. Xception [15] and NFTAPE [16] are examples of SWIFI tools.

The injection of realistic software faults (i.e., software bugs) has been absent from fault injection effort for a long time. First proposals were based on ad-hoc code mutations [17, 18] but more recent proposals allow the injection of representative software faults based on comprehensive field studies on the most common types of software bugs [19].

The use of fault injection techniques to assess security is actually a particular case of software fault injection, focused on the software faults that represent security vulnerabilities or may cause the system to fail in avoiding a security problem. Neves et. al. presented a tool (AJECT) focusing on the discovery of vulnerabilities on network servers, specifically on IMAP servers [9]. In their work the fault space is the binomial (attack, vulnerability) creating an intrusion that will cause an error and, possibly, a failure of the target system. To attack the target system they used predefined test classes of attacks and some sort of fuzzing.

In the industry, fuzzing techniques allied to the signature of known attacks and vulnerabilities are used to automate the penetration testing of web applications and web services. These tools, called Web Application Vulnerability Scanners, perform security testing and assessment, producing reports compliant with many security regulations (Sarbanes-Oxley, Payment Card Industry Data Security Standard compliance, etc.) that apply to web applications. Some of the best known are the HP WebInspect 7.7, the IBM Watchfire AppScan 7.0, the Acunetix web application security scanner, the WebSphinx, among others. In spite of their continuous development, these automated scanners still have some problems related to the high number of undetected vulnerabilities and high percentage of false positives [20].

Another contribution to better understand the most common vulnerabilities in web applications was presented in a field study that classified 655 Cross Site Scripting (XSS) and SQL Injection security patches of six widely used LAMP (Linux, Apache, MySQL and PHP) web applications [10]. One major conclusion of this study is that the most common type of vulnerabilities in web application code is by far, the "Missing Function Call – extended" (MFCE) that can be expanded into three sub-types (see Table 1). This fault type represents vulnerabilities caused by an input variable that should have been properly sanitized by a specific function (which the programmer forgot to include in the code). Table 1 shows that sub-type A, which is originated by unchecked numeric fields, is the most relevant. This result is also corroborated by another study, this time referring only to SQL injection vulnerabilities found in

Table 1. Missing Function Call - extended sub-types [10]

Sub-type	SQL (%)*	Description
A	64.25	Missing casting to numeric of one variable
B	4.15	Missing assignment of one variable to a custom made function
C	4.15	Missing assignment of one variable to a PHP predefined function

* The values are referred to all the SQL Injection vulnerabilities analyzed in the field study [10]

BugTraq SecurityFocus and presented by the Open Web Application Security Project (OWASP) [21]. This study concludes that about half of the SQL Injection vulnerabilities come from exploitation of numeric fields.

The methodology proposed in the present paper relies on the results of the already mentioned field studies [10, 21] to define the types of vulnerabilities to be injected (fault models), which match the most common types of vulnerabilities found in web applications in the field.

3. Vulnerability and attack injection approaches

In this paper we present a methodology that can be used to test important security mechanisms applied to web applications. The methodology is based on the injection of realistic vulnerabilities and subsequent controlled exploit of the vulnerabilities to attack the system. This provides a practical environment that can be used to test counter measure mechanisms (such as IDS, Web Application Vulnerability Scanners, Firewalls, etc.), train and evaluate security teams, estimate security measures (like the number of vulnerabilities present in the code), among others.

To provide a realistic environment we must deal with true to life vulnerabilities. As mentioned previously, we rely on the results from field studies [10, 21], particularly from the study presented in [10] that classified 655 security patches of six widely used LAMP web applications. With this data, we are able to define where a real vulnerability is usually located in the source code, what is the difference between a vulnerable and a non-vulnerable piece of code, and sometimes how the vulnerability manifests itself.

The injection is done by means of an automated tool: the **Vulnerability Injection tool**. The second element of the methodology, the attack of the vulnerability, is done by the **Attack Injection tool**. In fact, the Attack Injection tool also seamlessly integrates the Vulnerability Injection tool and both tools do their work as one, in an automated fashion.

The Vulnerability Injection tool is used to inject vulnerabilities in a web application source code file (Figure 1). It starts by analyzing the source code of the target file searching for locations where vulnerabilities can be injected. It follows the realistic patterns resulting

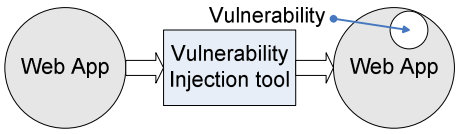

Figure 1: The Vulnerability Injection tool

from the field study [10]. Once it finds a possible location, it performs a specific code mutation in order to inject one vulnerability in that particular location. The change in the code follows the rules derived from [10], which are described and implemented by a set of Vulnerability Operators [11]. As a result, we obtain the original file with a single vulnerability injected. This procedure can be automatically repeated until all the locations where realistic vulnerabilities can be injected are identified and all the corresponding vulnerabilities are injected, resulting in a set of files, each one with one possible vulnerability.

The Vulnerability Operators are built upon a pair of attributes: the **Location Pattern** and the **Vulnerability Code Change**. The Location Pattern defines the conditions that a specific vulnerability type must comply with and the Vulnerability Code Change specifies the actions that must be performed to inject this vulnerability, depending on the environment where the vulnerability is going to be injected. In order to clarify the concept of Vulnerability Operator, let us analyze the following example [11]. One of the Location Pattern restrictions for the MFCE - A sub-type, is the search for the "intval()"[1] PHP function when the argument is related to an input value from the outside and the result is going to be used in a SQL query string. Consider, for example this code: "$id=intval($_GET['id']);". If the variable "$id" is going to be used in a query, then the Vulnerability Code Change consisted of removing this function from the source code in order to inject a vulnerability. As can be seen, by removing the function we get "$id=$_GET ['id';", which can be vulnerable to an SQL injection attack. For example, by assigning the value "15 or 1=1" to the "$id" variable, the SQL query is going to execute without the effect of the "where" condition, therefore affecting every row of the query, which was not intended by the developer of the application.

The automated attack of a web application with one vulnerability injected is done by the Attack Injection tool, in two stages. In the first stage, the web application is interacted (crawled) while both the HTTP and SQL communications are captured and processed by the tool. In the second stage (the attack stage, shown in Figure 2), a new interaction with the web application is performed but, this time, a collection of attack payloads is also applied in order to exploit the vulnerability by altering the SQL query sent to the database server of the web

[1] The "intval()" PHP function returns the integer value of the argument.

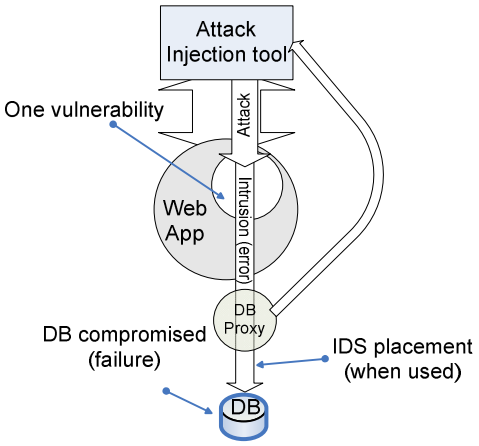

Figure 2: The attack stage of the Attack Injection tool

application. The interaction with the web application is always done from the web client's point of view (the web browser) and the payload is applied to the variables (the text fields, combo boxes, etc, present in the web page interface). At the end of the attack, the Attack Injection tool assesses if the attack was successful. This success is equivalent to the "error" state in a traditional fault injection technique. We are not looking for the "failure" that may result from the attack. If an "error" can be injected it means the attack was effective. Obviously, the consequences of the attack (the "failure") are dependent on the concrete situation and on how valuable is the data stored in the database. The detection of the success of the attack is done by searching for changes in the structure of the SQL query, showing the payload footprint.

To exemplify a possible use of the Attack Injection tool we can evaluate an SQL Injection IDS. In this case, the IDS is placed between the DB proxy and the database, as seen in Figure 2. During the attack stage, when the IDS inspects the SQL query sent to the database, the Attack Injection tool also monitors the output of the IDS to identify if the attack has been detected by the IDS or not. The entire process is performed automatically, without human intervention. We just have to collect the final results of the Attack Injection tool, which also contains, in this case, the IDS detection output.

In the next subsections, we describe the building blocks of the Vulnerability Injection and Attack Injection tools.

3.1 Vulnerability Injection Tool

The injection of vulnerabilities relies on data obtained by the field studies of vulnerabilities in LAMP web applications presented in [10, 21] and [11]. They refer to two of the most important vulnerability types in web applications, which are XSS and SQL Injection. An SQL Injection attack consists of tweaking the input fields of the web page (which can be visible or hidden) in order to alter the query sent to the backend database. This allows the attacker to retrieve sensible data or even alter database records. An SQL Injection attack can even be dormant for a while and be triggered by a specific event, such as the periodic execution of some procedures in the database (e.g., a scheduled database record cleaning function). A XSS attack consists of injecting HTML and/or a scripting language (usually Javascript) in a vulnerable web page. This attack type exploits the confidence a user has on the web site. The attack can affect other users of the web site, allowing the attacker to impersonate these users and even execute other types of attacks such as Cross Site Request Forgery (CSRF) [22]. The injection of XSS can also be persistent if the malicious string is stored in the backend database of the web application.

What both XSS and SQL Injection vulnerability types have in common is the fact that they are the result of poorly coded applications that do not check properly their inputs. There are many possible ways to prevent these vulnerabilities, but a field study concludes that a large majority of them (about 3/4) is through the use of a filtering function [10]. This missing function in a vulnerable web application was classified into the three sub-types shown in Table 1.

The list of possible types of vulnerabilities affecting web applications is enormous, but XSS and SQL Injection are at the top of that list, accounting for 32% of the vulnerabilities observed [1, 4]. Furthermore, SQL Injection is also responsible for some of the more severe results of attacks to web applications [6, 23, 24]. Nowadays, the most valuable asset of web applications is their back-end database. This is why the database is one of the main targets in web application attacks. For this reason, we have chosen to code first the SQL Injection type in our tools, although the XSS is quite similar in key aspects.

Figure 3 shows the main components of the Vulnerability Injection Tool. It comprises components to search for included files, analyze variables and finally inject vulnerabilities.

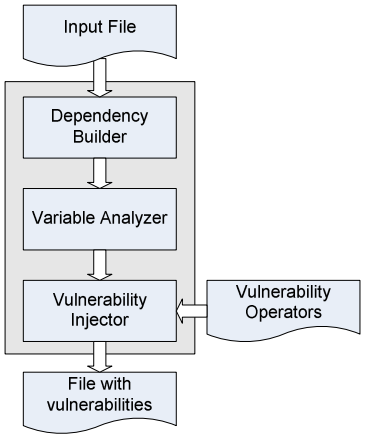

Figure 3: Architecture of the Vulnerability Injection tool

The first component of the Vulnerability Injection Tool is the **Dependency Builder**. It searches recursively for the files that are included in the **Input File**, which is the target PHP file where we want to inject the vulnerabilities. In PHP programming, it is common to include a generic file inside another file, for reutilization purposes (this is done using one of the following statements: *include()*, *include_once()*, *require()*, *require_once()* [25]), what happens in many other programming languages. When the web application is running, the execution of the original file and its included files is processed by the PHP interpreter as a single block of code. When searching for possible situations where vulnerabilities may be injected, one should analyze the code in the same way the PHP interpreter does, thus the inclusion of this Dependency Builder component.

The next component is the **Variable Analyzer**. Because XSS and SQL Injection vulnerabilities rely on vulnerable variables to be exploited, we have to analyze all the variables that influence SQL queries. This component gathers all the PHP variables from the source code and builds a mesh of dependencies related to each other. Then, it searches for PHP variables present in SQL query strings. Using the mesh created, the component can also determine all the variables that are indirectly responsible for the SQL query. Both variables that are directly and indirectly responsible for SQL Injection (or XSS, if it were the case) are considered valid. This is important because one variable may be used only as input (POST or GET HTTP parameters) and the result is passed to another variable that is the one that is going to be in the SQL query string. All the other variables are discarded.

Finally, the last component is the **Vulnerability Injector**. During the execution of this component, every location where the selected variables are found is tested with the conditions and restrictions of the **Vulnerability Operators** defined in [11], filtering those that are not applicable. The Vulnerability Injector component uses the Vulnerability Operator data and the result is the information about the mutation that has to be made in the source code in order to inject a particular vulnerability. Both the original source code and the mutated code (vulnerability injection code) are stored in the internal database of the Vulnerability Injection tool for future consumption (e.g. during the execution of the Attack Injection tool).

In addition to working as an element of the Attack Injection tool, the Vulnerability Injection tool can also be used standalone (e.g. for the training of security teams [11]) and the immediate generation of the PHP files with vulnerabilities is also a feature built into this component.

3.2 Attack Injection Tool

We see the Attack Injection tool as an all in one application: it injects vulnerabilities into a web application and attacks them in a seamlessly manner. Therefore, the Attack Injection tool has the Vulnerability Injection tool integrated as a building block (Figure 4). The process of attacking an application consists of three stages: the **Preparation**, the **Injection of Vulnerabilities** and the **Attack**. The Preparation and the Injection of vulnerabilities are executed side by side, producing a set of results that will be used by the last stage, the Attack.

During the **Preparation stage**, the web application is executed and the interaction is surveyed by the tool. This interaction can be made either manually, by someone executing every web application procedure, or automatically using an external tool, such as a **Web Application Crawler**, for example. During this interaction the HTTP communication protocol between the web browser and the web server and all the SQL communications going to and from the database server (MySQL in our prototype tool) are intercepted by the Attack Injection tool.

This interception is accomplished using built in proxies specifically developed for the HTTP and for the SQL communications. These proxies send the entire packets data traversing them through the configured ports to the Attack Injection tool components **HTTP Communication Analyzer** and **MySQL Communication Analyzer**. Because these proxies run as independent processes and threads, they are relatively autonomous and asynchronous. To guarantee that they are perfectly synchronized with other components of the Attack Injection tool, a **Sync** mechanism was also built in (Figure 4). This allows, for example, matching the input interaction with the respective proxies' results.

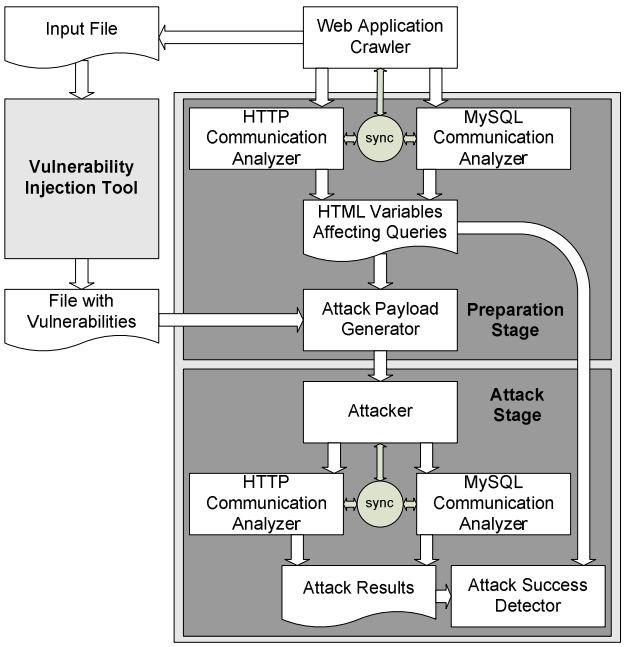

Figure 4: Architecture of the Attack Injection tool

The information gathered by both proxies allows obtaining the structure of each web page, the associated input variables, typical values, and the associated SQL queries. During this interaction, the list of the web application files that are being run is also sent to the integrated **Vulnerability Injection tool** as input files. For each one, the Vulnerability Injection tool is executed, delivering the respective group of files with vulnerabilities already injected. Each of these files has only one vulnerability injected, i.e. the protection of that particular variable has been removed from the source code, following the Vulnerability Operator rules.

Each one of these vulnerable variables must be attacked and for that purpose, the **Attack Payload Generator** creates a collection of attack payloads for every one of these files, according to characteristics of the target variable. These payloads intend to inject unwanted features in the queries sent to the database, therefore executing SQL Injection. They are based on a collection of basic attack strings presented in Table 2, but they may be extended, covering other cases, like those presented by Halfond et al. [26]. Every attack string is attached to the vulnerable variable trying to create some sort of text that can penetrate the breach produced by the injected vulnerability. Some tweaks are done to the payload strings, such as encode some parts using a URL encoding function. An attack footprint is also calculated that is expected to be seen in the query, if the attacked is successful. This is the last step of the Preparation stage and the data generated is prepared to be used by the attack stage of the Attack Injection tool.

The **attack stage** receives the files with vulnerabilities and the attack payloads from the previous stage. During the attack, every vulnerability is injected into the corresponding PHP file, one at a time. To prevent bias from previous attacks, before injecting any vulnerability, the web application files are copied from a safe location and the web application database is restored from a clean backup made before the start of the whole process. Using the generated attack payload, the web application is automatically attacked. While the attack is being performed, once again, the HTTP and SQL communications are intercepted by the respective proxies and results are analyzed and stored in the internal database by the **HTTP Communication Analyzer** and **MySQL Communication Analyzer**, as explained before.

Table 2. Basic attack payload string examples

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

After the end of the attack, it is necessary to verify if it was successful or not. This is done by the **Attack Success Detector** component. The attack is successful if, as a result of the execution of the payload, the structure of the SQL query is altered [27]. This occurs when the payload footprint shows up in the query in specific conditions. We do not consider the cases where the payload footprint is placed inside a string variable of the SQL query, because usually a string can convey any combination of characters, numbers and signs. In the other cases, if it is possible to alter the structure of the query, then we have a successful SQL Injection attack.

There is, however, one situation that can be misinterpreted by the Attack Injection tool. It occurs when the vulnerable variable value is processed by the web application code before being included in the SQL query. For example, if the input value is the full name of a person and the web application splits it into the name and surname; then the name and surname are going to be used in the SQL query in two different columns. This kind of processing cannot be detected correctly by the current implementation of the algorithm of the Attack Injection tool, therefore the attack payload footprint generated will be void. On the other hand, if the full name was used in the same query column the payload footprint will be fine. For this type of processing of the input variable, we have only implemented the situation where the processing done to the variable is changing the typesetter case of the value. Other common situations such as word separation, last name detection, etc, could also be easily implemented and added.

One final remark about the Attack Injection tool is that it does not try to exploit the vulnerability in the sense of obtaining sensible information from the web application database. It only tries to evaluate whether some particular instance of the web application (depending on the vulnerability injected) is vulnerable to such attacks or not. The Attack Injection tool also makes the attacker SQL query string and the specific vulnerability exploited available. The output information given by the Attack Injection tool is the most important outcome and it is also the most relevant data for enterprises. This allows developers of the tools under assessment to upgrade them and correct the weaknesses discovered.

4. Experimental results

To evaluate the Attack Injection tool (and the

integrated Vulnerability Injection tool) we have made three groups of experiments. In the first group, we injected vulnerabilities into three web applications to verify the quality of the vulnerabilities injected and the attack performance. In the second group, we tested an IDS for databases. The goal was to evaluate the efficiency of the IDS in detecting the attacks injected by the Attack Injection tool. In the final group of experiments, we evaluated two commercial Web Application Vulnerability Scanners regarding the detection of vulnerabilities that may be exploited for ad-hoc SQL Injection. In this situation, we tested the scanners with the vulnerabilities that could be attacked by the Attack Injector tool.

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 common technologies used to build custom web applications nowadays.

We used three web applications as setup for the experiments. The first is the groupware/content management system TikiWiki [28]. It allows building wiki (a web site allowing users to contribute to it by adding and modifying its contents). It is widely used for building sites, such as the Official Firefox Support site and the KDE wiki. It was one of the finalists of the sourceforge.net 2007 for the most collaborative project award.

The second web application used is the phpBB. It is a well-known LAMP web application and it has become the most widely used Open Source forum solution [29]. It is used by millions of users worldwide and won the sourceforge.net 2007 community choice awards for best project for communications. It is also the forum module that is integrated into the phpNuke content management and portal web application. For these two applications (TikiWiki and phpBB), in order to limit the quantity of data that we had to analyze, we bounded the attack surface only to the public sections.

The last web application used is custom made and is called MyReferences. It consists of 13 PHP files and it is used to manage publications: it allows the storage of PDF documents, and some information about them such as the title, the conference, the year of publication, the document type, the relevance, and the authors. The information may be edited, queried and displayed.

4.1 Vulnerabilities and attacks injected

For this first experiment, we want to validate the ability of the Attack Injection tool in injecting vulnerabilities and also in exploiting them to attack web applications. Although this process is mostly automatic, it consists of the Preparation Stage, Vulnerability Injection Stage and Attack Stage. The Vulnerability Injection Stage is executed during the Preparation Stage (Figure 5).

Step 3 of the process can be manual or use a web crawler. During the development of the Attack Injection tool we started by executing this stage interacting manually with the web application for internal testing and debugging purposes. However, in order to keep the same conditions for all the applications analyzed we did all the tests using a web crawler. There are several of them [30], but only some are able to insert values in the web application fields such as the WebSphinx. For this purpose, we can also use the crawlers built in the Web Application Vulnerability Scanners, which are usually very good in performing this task of web site exploration.

The results of the attack injection in our target web applications are depicted in Table 3. The vulnerabilities injected represent all the MFCE SQL Injection type of vulnerabilities that can realistically be injected into the files used in the experiments. We must recall that these vulnerabilities must comply with a restrictive set of rules in order to be considered realistic. On average, the tool could inject one vulnerability for every 129 lines of PHP code. Given the small number of web applications analyzed, this number is merely informative and cannot be generalized. The tool took about 11 minutes in the attack stage of the TikiWiki web application, 12 minutes for the phpBB and 4 minutes for the MyReferences.

For each vulnerability, we used several attack payloads and 38% of these attacks were successful. This measure of success comes from the presence of the payload footprint in the SQL queries sent to the database. For

1. Reset the MySQL database 2. Start the execution of the inspection daemons (the HTTP and MySQL proxies are deployed) 3. Execute the web application (either manual or using an automatic crawler) 4. Stop inspection 5. Process and store the information generated during the execution of the web application	Preparation
6. Inject the vulnerabilities executing the integrated Vulnerability Injection tool	(*)
7. Reset the MySQL database 8. Generate attack payloads	Stage
9. Start the execution of the inspection daemons (the HTTP and MySQL proxies are deployed) 10. Inject a vulnerability 11. Start attack by sending every payload 12. Reset the MySQL database 13. Restore PHP files 14. Repeat the attack stage until every vulnerabilities have been injected 15. Calculate the attack success 16. Store attack results	Attack Stage

(*) Vulnerability Injection Stage

Figure 5: Execution stages of the Attack Injection tool

Table 3. Web applications attack injection results

Web Apps	Files Attacked	Source Code Lines	Vuln. Injected	Attacks	Successful Attacks	Vuln. Attacked Successfully
TikiWiki	tiki-editpage.php	904	3	84	34	3
	tiki-index.php	648	1	7	6	1
	tiki-login.php	305	3	21	0	0
	Total	**1857**	**7**	**112**	**40**	**4**
phpBB	search.php	1405	3	42	42	3
	login.php	224	1	21	21	1
	viewforum.php	694	1	7	7	1
	viewtopic.php	1210	5	84	84	5
	posting.php	1106	4	112	112	4
	Total	**4639**	**14**	**266**	**266**	**14**
MyRefs	edit_paper.php	310	27	525	61	20
	edit_authors.php	169	6	196	46	5
	Total	**479**	**33**	**721**	**107**	**25**
	Grand total	**6975**	**54**	**1099**	**413**	**43**

future work, we intend to deeply analyze the distribution of the attack payloads in the success of the attacks. We also want to test with other payload strings not yet included in Table 2 (see [26]), in order to improve the attack success rate. However, the current attack payloads were successful in 80% of the vulnerabilities injected.

We analyzed, one by one, each injected vulnerability that was not successfully attacked, in order to understand the reason why the attack was not successful. In five situations of the *edit_authors.php* file belonging to the MyReferences web application the vulnerability is injected by removing an "intval()" PHP function. By removing this function it is expected that the variable could be attacked injecting string values, such as " or 1=1" (see Table 2 for more examples). However, the affected variables are used inside strings formatted with the "%d" format, which filters non-numeric variables. Therefore, this string formatting gives another level of protection preventing the attack through the target variable. In these situations, when the tool injects one vulnerability (by removing the code responsible for the sanitation of the variable) it leaves the other pieces of code still preventing the variable to be exploited. Recall that, even if multiple vulnerabilities can be injected in a file, we used only a single vulnerability injected at a time. This means that one variable with multiple overlapping checks cannot be attacked if one of the checks is missing, because the others are still active. It could be possible, however to inject more than one vulnerability at the same time in the same file, but we have no field study data that supports this kind of setup.

All the other situations where it was not possible to attack the vulnerability, including those that affect the *tiki-login.php* of the TikiWiki web application, are the result of a mistake made by the Attack Injection tool. By coincidentally, the same variable used in a query is also used as a GET parameter in a form to send some information to the following web page. The Attack Injection tool was tricked by this second use of the variable and tried to inject a vulnerability in this latter place, which is of no effect to the SQL query (placed before in the source code sequence).

The vulnerabilities that did not produce any successful attack represent only 11% of all the vulnerabilities injected. Except for the particular cases explained before, the results show that the tools are effective in providing a sufficient number of realistic vulnerabilities in a web application and that these vulnerabilities can be exploited to launch successful attacks.

4.2 Case study 1: IDS evaluation

One possible use for the Attack Injection tool is the evaluation of security counter measures, such as an IDS. In this situation, the IDS must be integrated with the Attack Injection tool, because the IDS output must be closely monitored during the attack stage, as was explained in Section 3.

For this case study, we used an IDS for databases [31]. It can deal with Oracle and MySQL databases, but we only needed the latter. This IDS implements the anomaly detection approach having itself a learning phase and a detection phase. Before the Attack Injection tool is initiated, we have to train the IDS with the target web application. For this phase, we used a web crawler to execute the web application functions.

After the training phase of the IDS the Attack Injection tool can be configured to monitor the IDS output and operate with it.

The results of these experiments for the three web applications are shown in Table 4. In this table, the most important columns are the last two: Successful Attacks and Detected IDS Attacks. As can be seen, this IDS could detect almost 99% of the attacks injected. Only five of them were not detected. We can also observe that, allied to the high detection rate of the IDS, there is also a high false positive rate.

The Attack Injection tool does not only provide the results show in the Table 4. It can also show the exact HTTP attack code, the payload used, the query sent to the database, etc. With all this information developers and security managers can improve their tools and procedures. In this case study, a defective function in the

Table 4. IDS evaluation results

Web Apps	Files Attacked	Vuln. Injected	Attacks	Successful Attacks	Detected IDS Attacks	IDS False Positives
TikiWiki	tiki-editpage.php	3	84	34	34	49
	tiki-index.php	1	7	6	6	1
	tiki-login.php	3	21	0	0	21
	Total	**7**	**112**	**40**	**40**	**71**
phpBB	search.php	3	42	42	42	0
	login.php	1	21	21	21	0
	viewforum.php	1	7	7	7	0
	viewtopic.php	5	84	84	84	0
	posting.php	4	112	112	112	0
	Total	**14**	**266**	**266**	**266**	**0**
MyRefs	edit_paper.php	27	525	61	61	294
	edit_authors.php	6	196	46	41	28
	Total	**33**	**721**	**107**	**102**	**322**
	Grand total	**54**	**1099**	**413**	**408**	**393**

IDS could easily be identified as responsible for the missing detections. In this function, there was one particular situation in processing the query structure that was not covered correctly.

4.3 Case study 2: Web Application Vulnerability Scanners evaluation

In this case study, we evaluate another type of security tool: Web Application Vulnerability Scanners. Web Application Vulnerability Scanners are commercial applications used to audit the web application security from the attacker's point of view: they try to penetrate the web application as a black box, without having access to the source code. These scanners provide an easy and 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. They can assess a myriad of security aspects such as XSS, SQL Injection, path traversal, file disclosure, web server vulnerabilities, etc. They use signatures of identified attacks to known web applications (and web application versions), but they can also test for ad-hoc XSS and SQL Injection. It is their ability to discover unreported SQL Injection vulnerabilities in web applications that we are going to test in this case study.

We used the HP WebInspect 7.7 [32] and the IBM Watchfire AppScan 7.0 [33] scanners, which we randomly named them Scanner 1 and Scanner 2. 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 way the experiments are executed with the scanners is different from the IDS. In this case, we executed the Attack Injection tool in advance for the three target web applications so we can inject the collection of vulnerabilities that can be attacked successfully. Then, for each one of these vulnerabilities, we tested with both scanners (one at a time) and collected the results. Before every execution of the scanners, we reset the web application database to prevent bias from previous experiments.

The results of the scanners test are shown in Table 5. The number of SQL Injection vulnerabilities detected by the scanners is minimal. They could only detect about 9% and 7% for the scanners 1 and 2, respectively. Scanners heavily rely on the output of the web application (what the web browser sees) to detect a vulnerability, but the way web applications are built nowadays allows them to hide most of their error messages, making the task to identify this type of vulnerabilities really difficult for these scanners. As a result, we cannot rely only on these tools to assess the security of an ad-hoc web application.

Although we did not test for XSS, we believe that the results would have been better, because many XSS issues are manifested in the client browser, where the scanner has access. However, for the second order type of XSS, where the attack string is stored in the database for latter

Table 5. Web application vulnerability scanners results

Web Apps	Files Attacked	Vuln. Injected	Vuln. Attacked Successfully	Vuln. Scanner 1	Vuln. Scanner 2
TikiWiki	tiki-editpage.php	3	3	1	0
	tiki-index.php	1	1	0	0
	tiki-login.php	3	0	0	0
	Total	**7**	**4**	**1**	**0**
phpBB	search.php	3	3	0	1
	login.php	1	1	0	0
	viewforum.php	1	1	1	0
	viewtopic.php	5	5	1	1
	posting.php	4	4	0	0
	Total	**14**	**14**	**2**	**2**
MyRefs	edit_paper.php	27	20	1	0
	edit_authors.php	6	5	0	1
	Total	**33**	**25**	**1**	**1**
	Grand total	**54**	**43**	**4**	**3**

deployment, the detection rate should also be along with the SQL Injection results.

To improve the detection rate for the SQL Injection vulnerabilities, the scanners could use a similar approach to ours with the Attack Injection tool, using a probe in the SQL communication path to gather useful data.

5. Conclusion

In this paper, we propose a method to automatically inject realistic vulnerabilities and attacks in web applications, and present a set of tools that implement the proposed approach. Our approach provides an effective way to assess and improve security mechanisms related to web applications.

We used a set of real-world case-studies to show the effectiveness of the tools to successfully inject and attack web applications, to test an IDS for SQL and to assess two commercial Web Application Vulnerability Scanners. The results show the effectiveness of the tools executing these assessment tasks. Furthermore, the results also lead to interesting conclusions about the mechanisms and tools under evaluation, showing that the tested IDS can detect practically all the attacks but with a high rate of false positives and that the vulnerability scanners do have difficulties in detecting most of the vulnerabilities injected.

References

[1] Christey, S., Martin, R., "Vulnerability Type Distributions in CVE", Mitre report, May, 2007

[2] Zanero, S., Carettoni, L., Zanchetta, M., "Automatic Detection of Web Application Security Flaws", Black Hat Briefings, 2005

[3] Jovanovic, N., Kruegel, C., Kirda, E., "Precise Alias Analysis for Static Detection of Web Application Vulnerabilities", IEEE Symp. on Security and Privacy, 2006

[4] Stock, A., Williams, J., Wichers, D., "OWASP top 10", OWASP Foundation, July, 2007

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

[6] Vnunet, August, 2007, http://www.vnunet.com/vnunet/news/2197408/monster-keptbreach-secret-five

[7] NTA, May, 2007, http://www.nta-monitor.com/posts/2007/05/annualsecurityreport.html

[8] Powell, D., Stroud, R., "Conceptual Model and Architecture of MAFTIA", Project MAFTIA, deliverable D21, 2003

[9] Neves, N., Antunes, J., Correia, M., Veríssimo, P., Neves R., "Using Attack Injection to Discover New Vulnerabilities", IEEE/IFIP International Conference on Dependable Systems and Networks, 2006

[10] Fonseca, J., Vieira, M., "Mapping Software Faults with Web Security Vulnerabilities", IEEE/IFIP Int. Conference on Dependable Systems and Networks, June 2008

[11] Fonseca, J., Vieira, M., Madeira, H., "Training Security Assurance Teams using Vulnerability Injection", IEEE Pacific Rim Dependable Computing conference, December 2008

[12] McConnell, S., "Gauging Software Readiness with Defect Tracking". Software, IEEE, 1997

[13] Arlat, J., Costes, A., Crouzet, Y., Laprie, J.-C., Powell, D., "Fault injection and dependability evaluation of fault-tolerant systems", IEEE Trans. on Computers, 42(8):913.923, August, 1993

[14] Iyer, R., "Experimental Evaluation", Special Issue FTCS-25 Silver Jubilee, IEEE Symp. on Fault Tolerant Computing, pp. 115-132, 1995

[15] Carreira, J., Madeira, H., Silva, J. G., "Xception: Software Fault Injection and Monitoring in Processor Functional Units", IEEE Trans. on Software Engineering, vol. 24, no. 2, February 1998

[16] Stott, D.T., Floering, B., Burke, D., Kalbarczpk, Z., Iyer, R.K., "NFTAPE: a framework for assessing dependability in distributed systems with lightweight fault injectors", Computer Performance and Dependability Symp., 2000

[17] Christmansson, J., Chillarege, R. "Generation of an Error Set that Emulates Software Faults". IEEE Fault Tolerant Computing Symp. – FCTS-26, 1996

[18] Madeira, H. Vieira, M., Costa, D. "On the Emulation of Software Faults by Software Fault Injection.", IEEE/IFIP Int. Conf. on Dependable System and Networks, 2000

[19] Durães, J., Madeira, H., "Emulation of Software Faults: A Field Data Study and a Practical Approach", IEEE Trans. on Software Engineering, Vol. 32, No. 11, November 2006

[20] Fonseca, J., Vieira, M., Madeira, H., "Testing and comparing web vulnerability scanning tools for SQL injection and XSS attacks", IEEE Pacific Rim International Symposium on Dependable Computing, December 2007

[21] Sam NG. CISA, CISSP. SQLBlock.com, www.owasp.org/images/7/7d/Advanced_Topics_on_SQL_Injection_Protection.ppt, 2006

[22] Cgisecurity.net, December 2008, http://www.cgisecurity.com/articles/csrf-faq.shtml#whatis

[23] SANS Institute, January, 2008, http://isc.sans.org/diary.html?storyid=3823

[24] Web Application Security Consortium, August, 2008, http://www.webappsec.org/lists/websecurity/archive/2008-08/msg00084.html

[25] The PHP Group, December, 2007, http://pt.php.net/

[26] Halfond, W., Viegas, J., Orso, A., "A Classification of SQL Injection Attacks and Countermeasures", Int. Symp. on Secure Software Engineering, 2006

[27] Buehrer, G., Weide, B., Sivilotti, P., "Using Parse Tree Validation to Prevent SQL Injection Attacks", International Workshop on Software Egineering and Middleware, 2005

[28] TikiWiki, December, 2008, http://tikiwiki.org/

[29] phpBB, December, 2008, http://www.phpbb.com/

[30] Java-source.net, 2008, http://java-source.net/open-source/crawlers

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

[32] SPI Dynamics Inc., May, 2008, http://www.spydynamics.com/products/webinspect/

[33] Watchfire Corporation, 2008, http://www.watchfire.com