Comparison of penetration testing tools for web applications

Frank van der Loo

Supervisor: Erik Poll

Research number: 653
Student number: 0314005

August 15, 2011

Executive summary

Testing the security of web applications with automated penetration testing tools produces relatively quick and easy results. However there are a lot of such tools, both commercial and free. In this thesis a selection of such tools are tested against a number of different test cases to compare the tools and find out the quality of such tools. For each test case the number of reported vulnerabilities by the tools is recorded per type of vulnerability. For each type of vulnerability the reported vulnerabilities are manually checked for false positives and false negatives.

The tools leave much to be desired. The tools appear to have problems with web applications that use techniques that are a bit more advanced than average pages, such as cookies for logging in or session ids. Further, the tools produce quite a lot of false positives and duplicate results. Also, all tools had false negatives. These false positives, duplicates and false negatives would have to be checked manually. This can take hours, especially for big web applications. Some of the tools also have problems with crawling a web application when techniques such as includes are used. Another problem is that the tools are mainly good in finding SQL injection and XSS, while other vulnerabilities are not always detected by every tool. Other problems of the tools are that they depend on the server for some vulnerabilities (mainly SQL injection) and fail detection of this vulnerability for certain servers. Some of the tools have their own specific problems that causes the tools to miss certain vulnerabilities.

Ultimately, it is impossible to name a tool that is the best. The usefulness of the tools depends on the web application that is going to be tested and the vulnerabilities that it is going to be tested for. After all none of the tools was the best for all types of vulnerabilities and for all test cases.

Contents

1 Introduction **6**

2 The vulnerabilities **8**

2.1 SQL injection 8
2.2 XPath injection 9
2.3 XSS 9
2.4 Cross site tracing 10
2.5 CSRF 10
2.6 Local file inclusion 10
2.7 Remote file inclusion 11
2.8 HTTP response splitting 11
2.9 Command injection 11
2.10 SSI injection 11
2.11 LDAP injection 12
2.12 Buffer overflow 12
2.13 Session management 12

3 Penetration testing tools **13**

3.1 Inner workings 13
3.2 Advantages 15
3.3 Limitations 15

4 The tools **16**

4.1 Commercial tools 16
4.1.1 HP WebInspect 16
4.1.2 JSky 16
4.2 Free/open source tools 17
4.2.1 w3af 17
4.2.2 Wapiti 17
4.2.3 Arachni 17
4.2.4 Websecurify 17
4.3 Summary 17
4.4 Tools not chosen 18
4.5 Tools chosen 18

5 Comparison **20**
- 5.1 Test setup . . . 20
- 5.2 Results . . . 22
  - 5.2.1 MiniSQLApp . . . 22
  - 5.2.2 WebGoat . . . 22
  - 5.2.3 phpBB . . . 29
  - 5.2.4 Mutillidae . . . 30
  - 5.2.5 zero.webappsecurity.com . . . 34

6 Related work **35**
- 6.1 General consensus . . . 35
- 6.2 Description of each paper . . . 35
- 6.3 Summary . . . 37

7 Conclusion **39**
- 7.1 Quality of the tools . . . 39
  - 7.1.1 Quality of the crawling . . . 39
  - 7.1.2 Quality of the fuzzing . . . 40
  - 7.1.3 Quality of the analyzing . . . 40
- 7.2 Conclusions per test case . . . 40
  - 7.2.1 General . . . 40
  - 7.2.2 MiniSQLApp . . . 41
  - 7.2.3 WebGoat . . . 41
  - 7.2.4 phpBB . . . 41
  - 7.2.5 Mutillidae . . . 41
- 7.3 Main conclusions . . . 41
- 7.4 Limitations . . . 42
- 7.5 Suggestions for improvement . . . 43
  - 7.5.1 Suggestions in general . . . 43
  - 7.5.2 w3af . . . 44
  - 7.5.3 Arachni . . . 44
- 7.6 Conclusion about each tool . . . 44
  - 7.6.1 General . . . 45
- 7.7 Choosing a tool . . . 45

8 Future work **46**

A MiniSQLApp **49**

List of Figures

5.1 JSky's report 24
5.2 w3af's report 25
5.3 Wapiti's report 26
5.4 Arachni's report 27
5.5 Websecurify's report 28

Chapter 1

Introduction

Testing the security of web applications is very important. There are several ways to do this. One such way is by using penetration testing tools. These tools test the security by performing an attack, without malicious payload (i.e. they will not delete parts of the web application or the database it uses), against the web application that should be tested. The results of these attacks are monitored by the tool to see which succeed.

There are a lot of penetration testing tools available, with different qualities.
Research question: Which penetration testing tool is the best?
Subquestions

1. What penetration testing tools exist?
2. How do these tools work?
3. What metrics can be used to compare the tools?
4. What test cases can be used to compare the tools?
5. What vulnerabilities can these tools detect and what not?
6. How do these tools compare in a test?

The tools will be tested by running them against a number of web applications and comparing the performance of the tools. The performance will be measured by counting the number of false positives and false negatives.

In chapter 2 the vulnerabilities the tools can test for will be explained. It will contain, per vulnerability, a description of the vulnerability, an explanation of the different form of the vulnerability (if any) and how a tool can test for the vulnerability. This chapter will answer subquestion 5.

Chapter 3 will go deeper into penetration testing tools, explaining the inner workings of such tools and the advantages and disadvantages. This chapter will answer subquestion 2 and partly subquestion 5.

In chapter 4 the tools chosen for this thesis will be introduced. This chapter will also explain why these tools were chosen and which tools were not chosen and why. This chapter also partly answers subquestion 1.

Chapter 5 contains the comparison. It will explain the test setup and the results per test case. This chapter answers subquestion 4 and 6.

In chapter 6 related work about penetration testing tools will be discussed. This will contain both the general consensus and conclusions drawn by individual papers. This related work will be used partly to answer subquestion 1 and 3.

In chapter 7 the final conclusions of this thesis are drawn. It will contain conclusions about the quality of the tools, suggestion for improvement of the tools and a section explaining how to choose a suitable tools for a certain web application. This chapter answers subquestion 6.

Chapter 8 contains ideas for future work in this subject.

Chapter 2

The vulnerabilities

This section lists the vulnerabilities the tools can test for according to the claims on the tools' website. For each vulnerability a description will be given, followed by the different types (if there is more than one type) and it will be explained how a tool can find the vulnerability. In this thesis only the first nine vulnerabilities will be used for testing the tools.

2.1 SQL injection

SQL injection (CWE-89 - Improper Neutralization of Special Elements used in an SQL Command) can occur when unvalidated user input is used to construct an SQL query that is then executed by the web server. A very well known example is a query used by a user login. This query is usually like "SELECT * FROM users WHERE username='*entered username*' AND password='*entered password*' ". If an attacker enters the string *x' OR '1'='1* in both the username and the password field the query becomes "SELECT * FROM users WHERE username='x' OR '1'='1' AND password='x' OR '1'='1' ". Because '1' is always equal to '1', this query is true for all records in the database. A more detailed description can be found at the OWASP website[1] and in [1].

There are two different types of SQL injection: blind SQL injection and "normal" SQL injection. The difference between these two types is that for "normal" SQL injection the server shows an error message when the SQL query's syntax in incorrect, for blind SQL injection this error message is not shown. Instead the attacker will see a generic error message or page.

"Normal" SQL injection can be tested for by entering characters like quotes to create a query with an incorrect syntax and search the page for error messages about it. Blind SQL injection can not be detected this way, instead the attacker has to enter SQL commands like *sleep* or statements that are always true or false. For instance trying both strings *' AND '1'='1* and *' AND '1'='2* will likely produce different results if the page is vulnerable to SQL injection.

[1] https://www.owasp.org/index.php/SQL_Injection

2.2 XPath injection

XPath injection (CWE-643 - Improper Neutralization of Data within XPath Expressions and CWE-91 - XML Injection (aka Blind XPath Injection)) is similar to SQL injection. The difference between these two vulnerabilities is that SQL injection takes place in a SQL database, whereas XPath injection takes place in an XML file as XPath is a query language for XML data. Just like SQL injection the attack is based on sending malformed information to the web application. This way the attacker can discover how the XML data is structured or access data he is not allowed to. More information about this attack can be found at the OWASP website[2].

Just like SQL injection, there are two types of XPath injection: "normal" XPath injection and blind XPath injection. The difference between these two types of XPath injection is that for blind XPath injection the attacker has no knowledge about the structure of the XML document and the application does not provide useful error messages.

Testing for XPath injection is also similar to SQL injection. The first step would be to insert a quote in an input field to see if it produces an error message. For blind XPath injection data is injected to create a query that always produces true or false.

2.3 XSS

Cross-site scripting (CWE-79 - Improper Neutralization of Input During Web Page Generation), often abbreviated as XSS. In short, it occurs when an attacker can input HTML code (such as Javascript), that will then be executed for the visitors of the site. An example would be a guest book that shows the text that is entered in the guest book on the website. If an attacker enters the string *<script>alert('XSS');</script>* a pop-up with the text "XSS" would be shown on that page of the guest book. This type of vulnerability can also be exploited in a more serious way. An attackers might use XSS to steal a user's cookie, which can then be used to impersonate the user on a website. A more detailed description can be found at the OWASP website[3] and in [1].

There are three different types of XSS: stored XSS, reflected XSS and DOM based XSS. The differences between these types are that, for stored XSS the attacker's code is stored on the web server (e.g. the guest book of the example above), whereas for reflected XSS the attacker's code is added to a link to the web application (e.g. in a GET parameter) and the attacker has to trick a user into clicking on the link. Such a link would look like *http://www.example.com/index.php?input=<script>alert('XSS');</script>*. For Dom based XSS the attacker's code is not injected in the web application, instead the attacker uses existing Javascript code on the target page to write text (e.g. *<script>alert('XSS');</script>* on the page.

To test for this vulnerability, a penetration testing tool should try to input HTML code in the inputs on a web application. After this, the tools would have to search for the code that was inputted, to see if it is present.

[2]https://www.owasp.org/index.php/XPATH_Injection
[3]https://www.owasp.org/index.php/Cross-site_Scripting_%28XSS%29

2.4 Cross site tracing

Cross Site Tracing often abbreviated as XST is an attack that abuses the HTTP TRACE function. This function can be used to test web applications as the web server replies the same data that is sent to it via the TRACE command. An attacker can trick the web application in sending its normal headers via the TRACE command. This allows the attacker to be able to read information in the header such as a cookie. A more detailed description can be found at the OWASP website[4].

To test for this vulnerability, the penetration testing tool will have to request OPTIONS from the web server and see if the headers Allow: TRACE are present. If that is present, the tool should try to request the page via the TRACE command.

2.5 CSRF

Cross-Site Request Forgery (CWE-352 - Cross-Site Request Forgery), often abbreviated as CSRF, is an attack where an attacker tricks a user's browser into loading a request that performs an action on a web application that user is currently authenticated to. For example an attacker might post the following HTML on a website or send it in an HTML email
<img src="http://www.bank.com/transfer_money?amount=10000&target_account=12345">.
If the user is authenticated at his bank website (at http://www.bank.com) when this link is loaded it would transfer 10000 from the user's account to bank account number 12345. A more detailed description can be found at the OWASP website[5] and in [1].

Testing for this attack is pretty similar to testing for XSS, the tool will have to check if it can inject a link that may have effect on an other web application (e.g. the link of the example) into the web application that is being tested.

2.6 Local file inclusion

Local file inclusion, also known as path traversal or directory traversal (CWE-22: Improper Limitation of a Path name to a Restricted Directory ('Path Traversal')), means that a file on the same server as the one where the web application is running is included on the page. A common example would be a web application with the URL *http://www.example.com/index.php?file=some_file.txt*, by manipulating the file parameter the attacker might be able to load a file that he should not be able to see. A more detailed description can be found at the OWASP website[6].

A tool can test for this vulnerability by entering a path to a local file (usually /etc/passwd) in an input or GET parameter. This can be done with an absolute path (e.g. /etc/passwd) or a relative path (e.g. ../../../../../etc/passwd). The tool will then have to check if the contents of the local file are present on the page.

[4] https://www.owasp.org/index.php/Cross_Site_Tracing
[5] https://www.owasp.org/index.php/Cross-Site_Request_Forgery_%28CSRF%29
[6] https://www.owasp.org/index.php/Path_Traversal

2.7 Remote file inclusion

Remote file inclusion is equal to local file inclusion, except for that the file that is included is a file from a different server than the one the web application is running on. An example of this vulnerability is the same as for local file inclusion. However, instead of changing the file name parameter to a local file, the attacker should enter a path to a remote file.

Testing for this vulnerability is also similar to local file inclusion. However, instead of a path to a local file a path to a remote file should be used (e.g. *http://www.something.com/index.html*).

2.8 HTTP response splitting

HTTP Response Splitting (CWE-113 - Improper Neutralization of CRLF Sequences in HTTP Headers), also known as CRLF is an attack where the attacker can control the data that is used in an HTTP response header and enters a newline in this data. For example, if a web application uses a redirect via a GET parameter (e.g. *http://www.example.com/index.php?page=somepage.html*), this redirect is sent via the HTTP headers to the browser (the "Location" header). An attacker can append a newline to the value of the GET parameter and add his own headers. This way an attacker can add a "normal" response header and can cause text to appear on the web application this way. A more detailed description can be found at the OWASP website[7] and in [1].

A penetration testing tool would have to enter a newline, followed by a HTTP header, in inputs that may be present in the HTTP response header. The tool will then have to check whether the server returns the data that is inputted in the header by the attacker.

2.9 Command injection

Command injection (CWE-77: Improper Neutralization of Special Elements used in a Command ('Command Injection')) means that the attacker can execute a command on the server. An example would be a web application that lets the user enter an IP address that the server will then send a ping to. If an attacker would enter the string *1.2.3.4;ls* the server would send a ping to the IP address 1.2.3.4 and run the command "ls". A more detailed description can be found at the OWASP website[8].

This vulnerability can be tested by a penetration testing tool by entering a semicolon followed by a command (e.g. "ls") into an input field that may be vulnerable and checking if the response of the web application contains the output of the injected command.

2.10 SSI injection

Server-Side Includes Injection (CWE-97 - Improper Neutralization of Server-Side Includes (SSI) Within a Web Page), often abbreviated to SSI injection

[7] https://www.owasp.org/index.php/HTTP_Response_Splitting
[8] https://www.owasp.org/index.php/Command_Injection

is an attack where the attacker can enter SSI directives (e.g. *<!–#include file="file.txt" –>* or *<!–#exec cmd="ls -l" –>*) that are then executed by the web server. A more detailed description can be found at the OWASP website[9].

To test for SSI injection a penetration testing tool would have to enter SSI directives in the inputs on a web application and see if the web server executes these by searching the web page for results of the SSI directive.

2.11 LDAP injection

LDAP injection (CWE-90 - Improper Neutralization of Special Elements used in an LDAP Query) is an attack where the attacker inputs LDAP statements that are executed by the server. More information can be found at the OWASP website[10].

There are two types of LDAP injection: "normal" LDAP injection and blind LDAP injection. Just like with SQL injection and XPath injection, the difference between these two types is that with blind LDAP injection no error messages are shown.

To test for LDAP injection a penetration testing tool should enter (parts of) LDAP statements in inputs for normal LDAP injection. For blind LDAP injection true or false questions should be entered in the inputs.

2.12 Buffer overflow

In short, a buffer overflow occurs when an application tries to store more data in a buffer than the buffer can hold. A more detailed description can be found at the OWASP website[11].

Testing for buffer overflows is relatively easy. The tool will have to input long (random) data and see if it produces any errors caused by trying to store more data than fits in the buffer.

2.13 Session management

Session management vulnerabilities can mean several things: session prediction[12], session fixation[13] or session hijacking[14]. It is not yet clear to me for which of the vulnerabilities the tools test for.

[9]https://www.owasp.org/index.php/Server-Side_Includes_%28SSI%29_Injection
[10]https://www.owasp.org/index.php/LDAP_injection
[11]https://www.owasp.org/index.php/Buffer_Overflow
[12]https://www.owasp.org/index.php/Session_Prediction
[13]https://www.owasp.org/index.php/Session_fixation
[14]https://www.owasp.org/index.php/Session_hijacking_attack

Chapter 3

Penetration testing tools

This section will be about penetration testing tools. Here, the inner workings of these tools are explained. Later in this section the advantages and limitations of penetration testing tools will be explained.

3.1 Inner workings

Most of the penetration testing tools use a technique that is called *fuzz testing*, *fuzzing* or *fault injection*. Fuzzing has been defined by [2] as:

> A highly automated testing technique that covers numerous boundary cases using invalid data (from files, network protocols, API calls, and other targets) as application input to better ensure the absence of exploitable vulnerabilities. From modem applications' tendency to fail due to random input caused by line noise on "fuzzy" telephone lines.

The part of the program that does this is called a *fuzzer* or a *fault injector*. Fuzzers can be divided in two main types:

Fuzzers that act as a proxy The user will have to set up his browser to use a proxy: the fuzzer. These tools monitor the web application the user visits and the data the user enters in input fields to find inputs [3].

Fuzzer that can crawl These tools will crawl the web application to find inputs that may be vulnerable and only rely on the user to provide a URL to crawl.

Fuzzers can also be divided in two types by a different property:

Generation fuzzers A fuzzer that generates its own data.

Mutation fuzzers A fuzzer that uses valid data, mutates this and replays the mutated data.

The usual steps that are performed by penetration testing tools to discover vulnerabilities are explained in [4].

1. Identify the target

2. Identify inputs
3. Generate fuzzed data
4. Execute fuzzed data
5. Monitor for exceptions
6. Determine exploitability

1. Identify the target Identifying the target consists of choosing the target to run the tool against.

2. Identify inputs Identifying the inputs consists of finding vulnerable spots. In a web application these spots are found by crawling the site to find links, action attributes of forms and the source attributes of other tags [5].

3. Generate fuzzed data After identifying the inputs, the fuzz data to send to these inputs has to be generated. Whether this data is generated in advance or generated dynamically depends on the target and data format.

4. Execute fuzzed data This step is where the actual penetration testing is performed with the fuzzer. It sends input to the vulnerable spots found when identifying the inputs.

5. Monitor for exceptions In this phase the result of every fuzz packet is monitored. This is important, because it is vital to know what packet caused what result (i.e. what vulnerability was discovered by what packet).

How this monitoring takes place depends on the target and the type of vulnerability that is tested. For instance, monitoring for XSS vulnerabilities will consist of crawling the web application again and searching for pages that contain the input that was injected during the execution of the fuzzed data [5]. Monitoring for SQL injection would consist of looking for SQL errors or other results that show that the injection was successful. These other results depend on the fuzzed data.

6. Determine exploitability This final phase is to determine whether a vulnerability discovered can be exploited. This process usually has to be done by a person rather than the tool.

An easier way to divide the steps a penetration testing tool performs is

Crawling The phase that crawls the web application to find the pages the web application consists of and vulnerable inputs. This corresponds with step 1 and 2 of the other division.

Fuzzing This phase sends the data to test the web application to the application. This corresponds with step 3 and 4 of the other division.

Analyzing In this phase the result of the fuzzing phase is analyzed to check if the web application is vulnerable. This corresponds with step 5 of the other division.

3.2 Advantages

The main advantage of using penetration testing tools it that it is a relatively fast and easy way to detect certain security vulnerabilities. Unlike traditional black box testing, in which an ethical hacker tries to attack the web application, penetration testing tools can be used by a person with little or no knowledge about security. Only the analysis of the result has to be done by a person with knowledge about security.

3.3 Limitations

Despite the advantages, penetration testing tools have limitations.

Some of these limitation are described in [6], [4] and [5]:

- Penetration testing tools can not find all vulnerabilities. Only vulnerabilities that cause results that can be monitored for. Therefore, they do a poor job at finding vulnerabilities like information disclosure and encryption flaws.
- Such a tool find cannot find access control flaws, identify hardcoded backdoors or identify a multivector attack (i.e. an attack that consists of chaining together multiple minor vulnerabilities which results in a major vulnerability) [4].
- For most applications random data will not uncover vulnerabilities. Even if it does, the fuzzing process will have to be repeated many times, because of its random nature [6].
- Automated penetration testing tools have no specific goals in mind to work towards. The tool, therefore, has to try to attack every possible risk [5].

Chapter 4

The tools

This section will introduce the penetration testing tools that will be used for this thesis. Here, the tools and the vulnerabilities they can detect will be described. This section will also explain why some of the tools that have been considered will not be used for this thesis.

Because tools that act as a proxy, as described in 3.1, rely on the user to guide them I have decided not to use such tools for this thesis. Only tools that can crawl the web application themselves will be used.

The tools that have been taken into consideration for this thesis were found in lists at various websites[1] [2] [3] [4].

4.1 Commercial tools

4.1.1 HP WebInspect

HP WebInspect is a commercial penetration testing tools from HP [7]. For this thesis the 15-day evaluation version will be used. The list of vulnerabilities it claims to test for can be found in its data sheet[5]. It appeared that this evaluation version can only be used against a web application on the web server of HP, therefore this penetration testing tool will only be used in one very limited test.

4.1.2 JSky

JSky is a commercial penetration testing tools from NOSEC [8]. For this thesis the 15-day fully functional evaluation version will be used. The list of vulnerabilities it claims to test for can be found on the tool's website[6].

[1] http://www.owasp.org/index.php/Phoenix/Tools7#RSnake.27s_XSS_cheat_sheet_based-tools.2C_webapp_fuzzing.2C_and_encoding_tools

[2] http://www.dragoslungu.com/2007/05/12/my-favorite-10-web-application-security-fuzzing-tools/

[3] http://www.securityprocedure.com/complete-list-free-web-application-security-scanner

[4] http://pentesttools.com/index.php/web-application-test-tools.html

[5] http://www.hp.com/hpinfo/newsroom/press_kits/2011/applicationtransformation/WebInspectDataSheet.pdf

[6] http://www.nosec-inc.com/en/products/jsky/

4.2 Free/open source tools

4.2.1 w3af

w3af is the abbreviation of the Web Application Attack and Audit Framework [9]. It is an open-source program, written in Python. It uses plugins to perform the attacks on the web application. A description of the vulnerabilities these plugins are claimed to detect can be found on the tool's website[7]. It uses a menu-driven text-based structure, but it also has a GUI. Results are outputted to the console or to an XML-, text-, or HTML-file.

4.2.2 Wapiti

Wapiti is another open-source program written in Python [10]. The vulnerabilities it claims it can detect can be found on the tool's website[8]. It works from the command-line completely automatically, however command-line options can be used to customize scanning. Output is written to the console or an XML-, text-, or HTML-file.

4.2.3 Arachni

Arachni is a Web Application Vulnerability Scanning Framework [11]. It is an open source program written in Ruby. It has a modular setup. A description of what the modules can test for can be found on the tool's website[9]. At the moment it only has a command-line interface. Running the program with as parameter a URL will automatically audit the web application on that URL with all modules. The audit can be customized with options on the command-line. The output can be sent to the console or a text-, XML-, HTML- or AFR (Arachni Framework Report)-file.

4.2.4 Websecurify

Websecurify is an open-source integrated web security testing environment [12]. A list of vulnerabilities it claims it can detect can be found on the tool's website[10]. It has a GUI interface and performs the testing automatically. Very few options can be controlled via settings.

4.3 Summary

The vulnerabilities that can be detected by at least two of the chosen tools are listed in table 4.1. An 'x' means that the tool in that columns claims to be able to detect the vulnerability in that row.

[7] http://w3af.sourceforge.net/plugin-descriptions.php
[8] http://wapiti.sourceforge.net/
[9] http://arachni.segfault.gr/overview/modules
[10] http://www.websecurify.com/overview

Table 4.1: Summary of common vulnerabilities the tools claim they can detect

	HP WebInspect	JSky	w3af	Wapiti	Arachni	Websecurify
XSS	x	x	x	x	x	x
SQL injection	x	x	x	x	x	x
CSRF	x		x		x	x
buffer overflow	x		x			
SSI injection	x		x			
remote file inclusion	x		x	x	x	x
local file inclusion	x		x	x		x
HTTP response splitting	x		x	x	x	
command injection	x		x	x	x	
session management	x					x
XPath injection			x	x	x	
LDAP injection			x	x	x	
Cross site tracing			x		x	

4.4 Tools not chosen

The tools that have been reviewed but were not chosen in this paper for a variety of reasons are listed in table 4.2. *Proxy* means that the tools acts as a proxy, *Not available* means that the program is no longer available for download, *Did not work* means that I was not able to get the program to detect a very simple SQL injection and *Other* means that the program was not chosen for a different reason, the reason is stated in the table.

4.5 Tools chosen

The commercial tool, JSky, was chosen because it was the only commercial tool that could be found that offered a fully functional trial period.

The free/open source tools were initially selected from lists on various websites and related work and based on the claims on the tool's website. After this initial selection the tools were tested against MiniSQLApp, a very simple test created by me, and only tools that worked against that test and could detect blind SQL injection were chosen. The only exception is Websecurify. That tool was chosen even though it was not able to detect blind SQL injection. This tool was still chosen because it was claimed on the website that the tool could detect the entire OWASP top 10.

Table 4.2: Tools not chosen

	Proxy	Not available	Did not work	Other
ProxMon	x			
Suru Web Proxy	x			
Paros	x			
WebScarab	x			
SPIKE	x			
XSSScan		x		
HTMangLe		x		
WebFuzz		x		
ASP Auditor		x		
WhiteAcid's XSS Assistant		x		
screamingCobra		x		
Overlong UTF		x		
Web Hack Control Center		x		
Web Text Converter		x		
Grabber			x	
Wfuzz			x	
WSTool			x	
Uber Web Security Scanner			x	
Grendel-Scan			x	
J-Baah			x	
JBroFuzz			x	
XSSFuzz				Only XSS
WSFuzzer				Only for SOAP services
RegFuzzer				Only for regular expressions
RFuzz				Only for HTTP requests
XSSFuzz				Only XSS
Acunetix WVS(Free version)				Only XSS
Falcove				Could not be installed
NeXpose community edition				Not for web applications

Chapter 5

Comparison

This section will be about the comparison of the penetration testing tools. First, the test setup will be explained and this section will end with the results of the comparison.

5.1 Test setup

Testing the tools consists of several tests. A small test created by me, MiniSQLApp, and four existing web applications: WebGoat, phpBB, Mutillidae and zero.webappsecurity.com. Unless otherwise specified all tests were performed with the tools set to scan for all vulnerabilities and otherwise the tools' default settings.

1. The first test, created by me, consists of three pages. One page with two links to the other pages. Each of the two other pages consists of one input field and one submit button. Both have the same SQL injection vulnerability, however one application shows an error message, the other one does not (blind SQL injection). This test was chosen as an initial quality test. The test case can be found in appendix A.

2. WebGoat is a very extensive web application consisting of approximately 69 vulnerable web pages, divided into 19 categories: General, Access Control Flaws, AJAX Security, Authentication Flaws, Buffer Overflows, Code Quality, Concurrency, Cross-Site Scripting, Denial of Service, Improper Error Handling, Injection Flaws, Insecure Communication, Insecure Configuration, Insecure Storage, Malicious Execution, Parameter Tampering, Session Management Flaws, Web Services and Challenge. Because WebGoat is meant to teach people about web application security, every page is called a lesson. This test case was chosen because of its extensiveness in number and type of vulnerabilities.

   Advantages of WebGoat are that it comes with its own Web Server: Apache Tomcat. Therefore it will produce the same results regardless of the computer it runs on. Also, every lesson has a link that will show the solution for that lesson, this can help to determine why a tool was not able to detect the vulnerability.

Disadvantages are that it is very large, therefore it takes quite a long time to run the tools against it. Further, a user has to authenticate with a username and password and a session id is stored in a cookie, this made it harder to run the tools against it.

3. phpBB is a well-known forum software. It is the only "real" web application in this test. It was chosen because of the fact that it is a "real" web application instead of one with vulnerabilities implemented on purpose.

   The advantage is that it has been used in the real world, so it is not equipped with vulnerabilities implemented on purpose. This provides a challenge for the tools, to see if they can find vulnerabilities that are harder to detect.

   The disadvantage is that not all vulnerabilities are known. There exist security advisories and changelogs for phpBB, but these not always pinpoint the exact location of a vulnerability and how this is exploited. Another disadvantage is that it requires a web server. Therefore the results of the tools depends partially on the web server that is used and its configuration, so the tools' results will not always be the same.

4. Mutillidae is similar to WebGoat in that it consists of several vulnerable web pages, divided into categories. It consists of 12 web pages, divided into 13 categories: the 10 categories of the 2010 OWASP top 10, malicious file execution, information leakage and improper error handling and denial of service. Multiple vulnerabilities exist in the pages. However, unlike WebGoat Mutillidae is a set of PHP scripts and requires a web server that is installed on the computer it is run on. This test case was chosen because it contains the entire 2010 OWASP top 10, but does not contain cases such as HTTP authentication like WebGoat.

   Advantages of Mutillidae are that it contains the entire OWASP top 10, i.e. the most import vulnerabilities. Also, every page contains hints to exploit the vulnerability. This makes it easier to find out what vulnerability is in that page, although WebGoat's approach with solution is easier. Finally, because it is a set of PHP scripts it is very easy to make changes.

   A disadvantage is that it requires a web server. Just like phpBB this may cause the tools' results to differ because of the web server that is used and its configuration.

5. zero.webappsecurity.com is an existing web application. It has been created by HP as an example for potential buyers of HP WebInspect. This web application is the only web application the free trial of HP WebInspect can test. It was chosen because it is the only way to make a comparison between HP WebInspect and the other tools, even though it may be biased in favor of HP WebInspect.

   Advantages of this web application is that it is running on a web server online from HP, so the web application will remain the same regardless of the computer the tools is installed on.

   Disadvantages are that it is not possible to find out how many and what vulnerabilities exist in the web application. Also, because this is the only

Table 5.1: Results of the simple SQL injection test on MiniSQLApp

	SQL injection	Blind SQL injection
JSky	yes	no
w3af	yes	yes
Wapiti	yes	yes
Arachni	yes	yes
Websecurify	yes	no

web application the trial version of HP WebInspect can test it may be optimized for that tool.

5.2 Results

5.2.1 MiniSQLApp

The first test is a small web application that I made, MiniSQLApp, with two SQL injection vulnerabilities, one "normal" and one blind. The results of this test can be found in table 5.1. All the tools, except for Websecurify and JSky, were able to find both vulnerabilities. The fact that JSky was unable to find the blind SQL injection is peculiar, as the tool has a module that should be able to find blind SQL injection vulnerabilities. However, the values it entered to find vulnerabilities are not suitable to find blind SQL injection. The reason Websecurify can not find the blind SQL injection vulnerability is the same as JSky's: the values it enters to find vulnerabilities are not suitable to find blind SQL injection vulnerabilities.

5.2.2 WebGoat

A different test was performed by running the tools against OWASP WebGoat[1] version 5.3-RC1. WebGoat is a web application with several vulnerabilities. It consists of several lessons and the goals for the user is to exploit the vulnerability to complete the lesson. Initially, the tools were not able to crawl WebGoat and find vulnerabilities. The cause of this was that WebGoat sets a cookie when it is first visited. The tools did not store this cookie and send it when visiting the other pages. This problem was solved by using a cookie from a normal browser. Websecurify has a built-in browser, that was used to set the cookie. JSky offers a similar functionality, however it still failed to authenticate despite using this functionality. The results of this test can be found in table 5.3, 5.4 and 5.5. The tables contain the total amount of vulnerabilities (i.e. vulnerable inputs) found, the number of true positives (TP), the number of false positives (FP) and the number of false negatives (FN). Numbers between brackets are the total numbers including duplicates. The runtime (in hh:mm:ss) of the tools against WebGoat can be found in table 5.2. For XST it could not be determined if the reported XST vulnerabilities are true or false positives, because the tools only test for XST by testing if the TRACE method is supported by the web server. The true and false positives have been determined manually. This was done mainly by entering the string as reported by the tools in the vulnerable

[1] https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project

input. Examples of these reports can be found in figure 5.1, 5.2, 5.3, 5.4 and 5.5.However, for some reported vulnerabilities (e.g. XSS injection in the value of the submit button) it was immediately clear that it was a false positive. Other true and false positives could be determined by comparing the reported vulnerabilities to reports of the other tools that have been checked for true and false positives already.

Peculiar about the results is that barely any tool discovered the SQL injection vulnerabilities. The reason the "normal" (i.e. non-blind) SQL injection was not discovered by most tools is because of the SQL error message WebGoat produces. All tools have a list of error messages or regular expressions of error messages produced by several SQL servers. However, only the list of Arachni contains the error message the SQL server of WebGoat produces. The reason Arachni did not discover all SQL injections is because it adds the string '–;' to the default value, or 1'–;' if there is no default value. For most SQL servers this will cause an error, however with the SQL server of WebGoat this may result in a valid query (if the value of the input is used between quotes in the query, appending the abovementioned string will result in a valid query).

Another peculiar result is the amount of XSS vulnerabilities that were discovered by the tools. The high number of false positives is caused because none of the tools is able to execute Javascript as explained below. The tools test for XSS vulnerabilities by entering HTML code into potential vulnerable spots and check whether or not the HTML that was injected is present on the page. However, if the injected HTML is present in an input box, the tool will count it as an XSS vulnerability even though code inside such a box will never be executed. So, even though the injected HTML is present in the page, it can not be exploited. Another cause of the number of false positives is the amount of input fields on the pages of WebGoat. Some pages have a lot of input fields, when all input fields "remember" the value that was entered there, the tool will count every field as an XSS vulnerability and every field could cause a false positive as explained above. Another reason for the high number of false positives, mainly of Wapiti, is the "Show Params" link in WebGoat. When clicked, this link causes all parameters (both GET and POST) to show on the page. This is the reason that Wapiti (and possible other tools) can find the HTML code that they tried to inject by setting it as the value for the submit button on the page.

The number of false positives of XPath injections reported by w3af can not be determined. The tool only reported that it found XPath injections, but it did not report on what page.

It was hard to determine the results of Websecurify. In WebGoat every page has two GET parameters: *screen* and *menu*. In the report of Websecurify the value of *screen* was wrong. It pointed to a non-existent page. The correct page had to be deduced by looking at the inputs in the report.

It is difficult to determine the total number of vulnerabilities in WebGoat as it appeared that some pages had more than one vulnerability. For instance, several of the pages with an SQL injection vulnerability also has an XSS vulnerability. Therefore the only way to count the total number of vulnerabilities in WebGoat is to try and count every vulnerability on every page. However, this method costs a lot of time. Therefore, instead, other vulnerabilities (i.e. XSS vulnerabilities in pages with on purpose implemented SQL injection vulnerabilities) will be ignored. The numbers in the tables only show the number of detected vulnerabilities in the pages with that type of vulnerability.

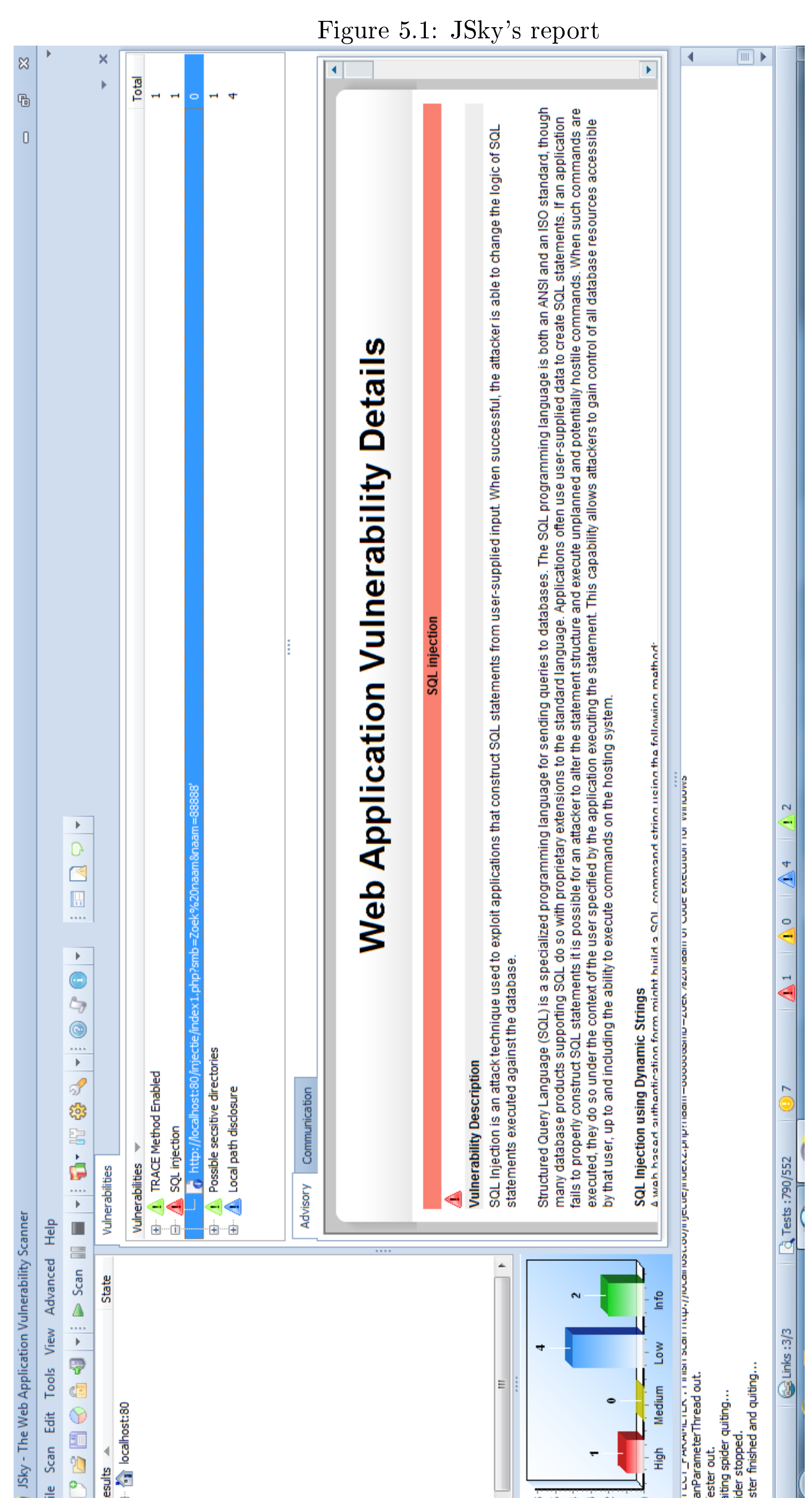


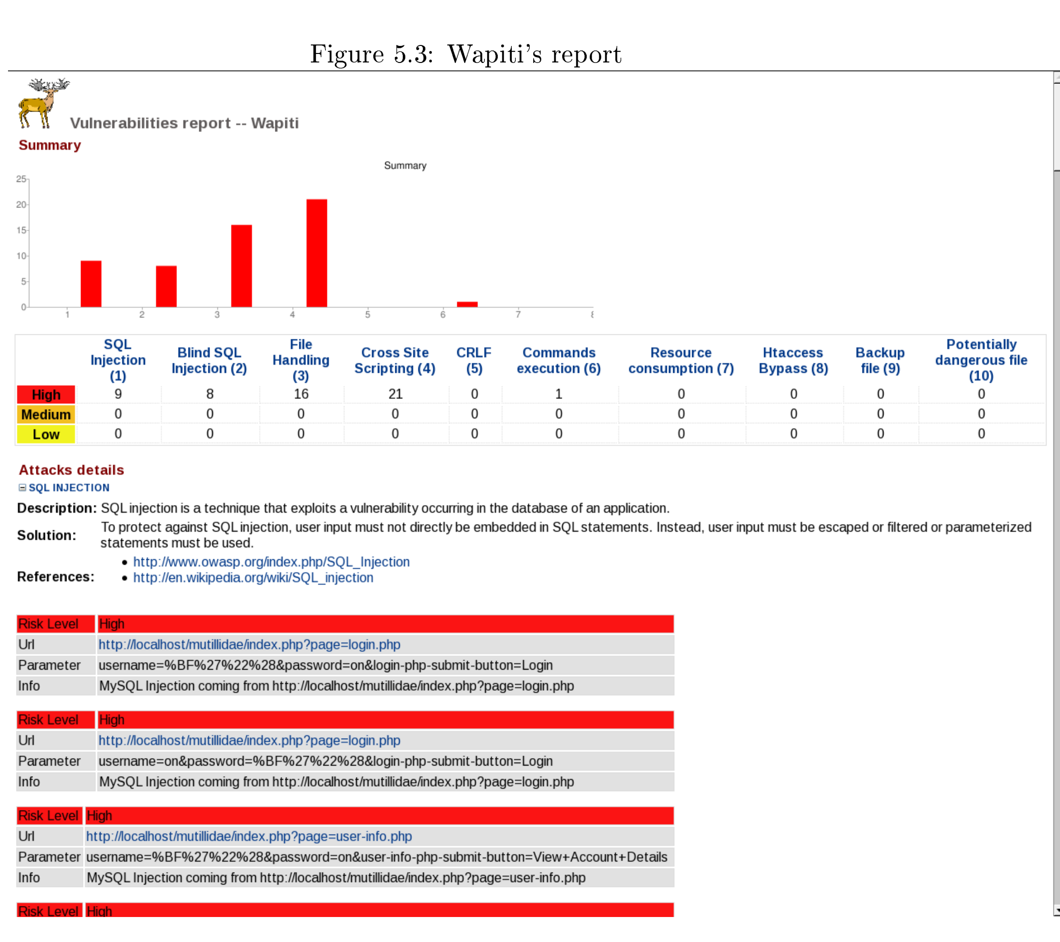
Figure 5.1: JSky's report

Figure 5.2: w3af's report

w3af target URL's
URL
http://localhost/mutillidae/index.php?page=login.php
http://localhost/mutillidae/index.php?page=user-info.php
http://localhost/mutillidae/index.php?page=dns-lookup.php
http://localhost/mutillidae /index.php?page=register.php
http://localhost/mutillidae/index.php
http://localhost/mutillidae/index.php?page=text-file-viewer.php
http://localhost/mutillidae/index.php?page=add-to-your-blog.php
http://localhost/mutillidae/index.php?page=view-someones-blog.php
http://localhost/mutillidae/index.php?page=browser-info.php
http://localhost/mutillidae/index.php?page=show-log.php
http://localhost/mutillidae/index.php?page=source-viewer.php
http://localhost/mutillidae /index.php?page=credits.php

		Security Issues
Type	**Port**	**Issue**
Vulnerability	tcp/80	Cross Site Scripting was found at: "http://localhost/mutillidae/index.php", using HTTP method GET. The sent data was: "page=<SCrIPT>alert("3Cp0")</SCrIPT>". This vulnerability affects ALL browsers. This vulnerability was found in the request with id 2287. **URL :** http://localhost/mutillidae/index.php Severity : Medium
Vulnerability	tcp/80	Cross Site Scripting was found at: "http://localhost/mutillidae/index.php?page=dns-lookup.php", using HTTP method POST. The sent post-data was: "dns-lookup-php-submit-button=Lookup+DNS&target_host= <ScRIPT>a=/RbPS/%0Aalert(a.source)</SCRiPT>". This vulnerability affects ALL browsers. This vulnerability was found in the request with id 2333. **URL :** http://localhost/mutillidae/index.php Severity : Medium
Vulnerability	tcp/80	Cross Site Scripting was found at: "http://localhost/mutillidae/index.php?page=register.php", using HTTP method POST. The sent post-data was: "username=<ScRIPT>a=/4pw2/%0Aalert(a.source)</SCRiPT>& confirm_password=FrAmE30.&password=FrAmE30.®ister-php-submit-button=Create+Account". The modified parameter was "username". This vulnerability affects ALL browsers. This vulnerability was found in the request with id 2358. **URL :** http://localhost/mutillidae/index.php

Figure 5.3: Wapiti's report



Vulnerabilities report -- Wapiti

Summary

	SQL Injection (1)	Blind SQL Injection (2)	File Handling (3)	Cross Site Scripting (4)	CRLF (5)	Commands execution (6)	Resource consumption (7)	Htaccess Bypass (8)	Backup file (9)	Potentially dangerous file (10)
High	9	8	16	21	0	1	0	0	0	0
Medium	0	0	0	0	0	0	0	0	0	0
Low	0	0	0	0	0	0	0	0	0	0

Attacks details

SQL INJECTION

Description: SQL injection is a technique that exploits a vulnerability occurring in the database of an application.

Solution: To protect against SQL injection, user input must not directly be embedded in SQL statements. Instead, user input must be escaped or filtered or parameterized statements must be used.

References:
- http://www.owasp.org/index.php/SQL_Injection
- http://en.wikipedia.org/wiki/SQL_injection

Risk Level	High
Url	http://localhost/mutillidae/index.php?page=login.php
Parameter	username=%BF%27%22%28&password=on&login-php-submit-button=Login
Info	MySQL Injection coming from http://localhost/mutillidae/index.php?page=login.php

Risk Level	High
Url	http://localhost/mutillidae/index.php?page=login.php
Parameter	username=on&password=%BF%27%22%28&login-php-submit-button=Login
Info	MySQL Injection coming from http://localhost/mutillidae/index.php?page=login.php

Risk Level	High
Url	http://localhost/mutillidae/index.php?page=user-info.php
Parameter	username=%BF%27%22%28&password=on&user-info-php-submit-button=View+Account+Details
Info	MySQL Injection coming from http://localhost/mutillidae/index.php?page=user-info.php

Risk Level	High

Figure 5.4: Arachni's report

```
[+] SQL Injection
[~] ~~~~~~~~~~~~~~~~~~~~
[~] Severity: High
[~] URL:      http://localhost:8080/webgoat/attack
[~] Elements: form
[~] Variable: station
[~] Description:
[~] SQL code can be injected into the web application.

[~] CWE: http://cwe.mitre.org/data/definitions/89.html

[~] Requires manual verification?: false

[~] References:
[~]   UnixWiz - http://unixwiz.net/techtips/sql-injection.html
[~]   Wikipedia - http://en.wikipedia.org/wiki/SQL_injection
[~]   SecuriTeam - http://www.securiteam.com/securityreviews/5DP0N1P76E.html
[~]   OWASP - http://www.owasp.org/index.php/SQL_Injection

[*] Variations
[~] ----------
[~] Variation 1:
[~] URL: http://localhost:8080/webgoat/attack?Screen=75&menu=1200
[~] ID:
[~] Injected value:     101'--;`
[~] Regular expression: Unexpected end of command in statement
[~] Matched string:     Unexpected end of command in statement


[+] Cross-Site Scripting (XSS)
[~] ~~~~~~~~~~~~~~~~~~~~
[~] Severity: High
[~] URL:      http://localhost:8080/webgoat/attack
[~] Elements: form
[~] Variable: password
```

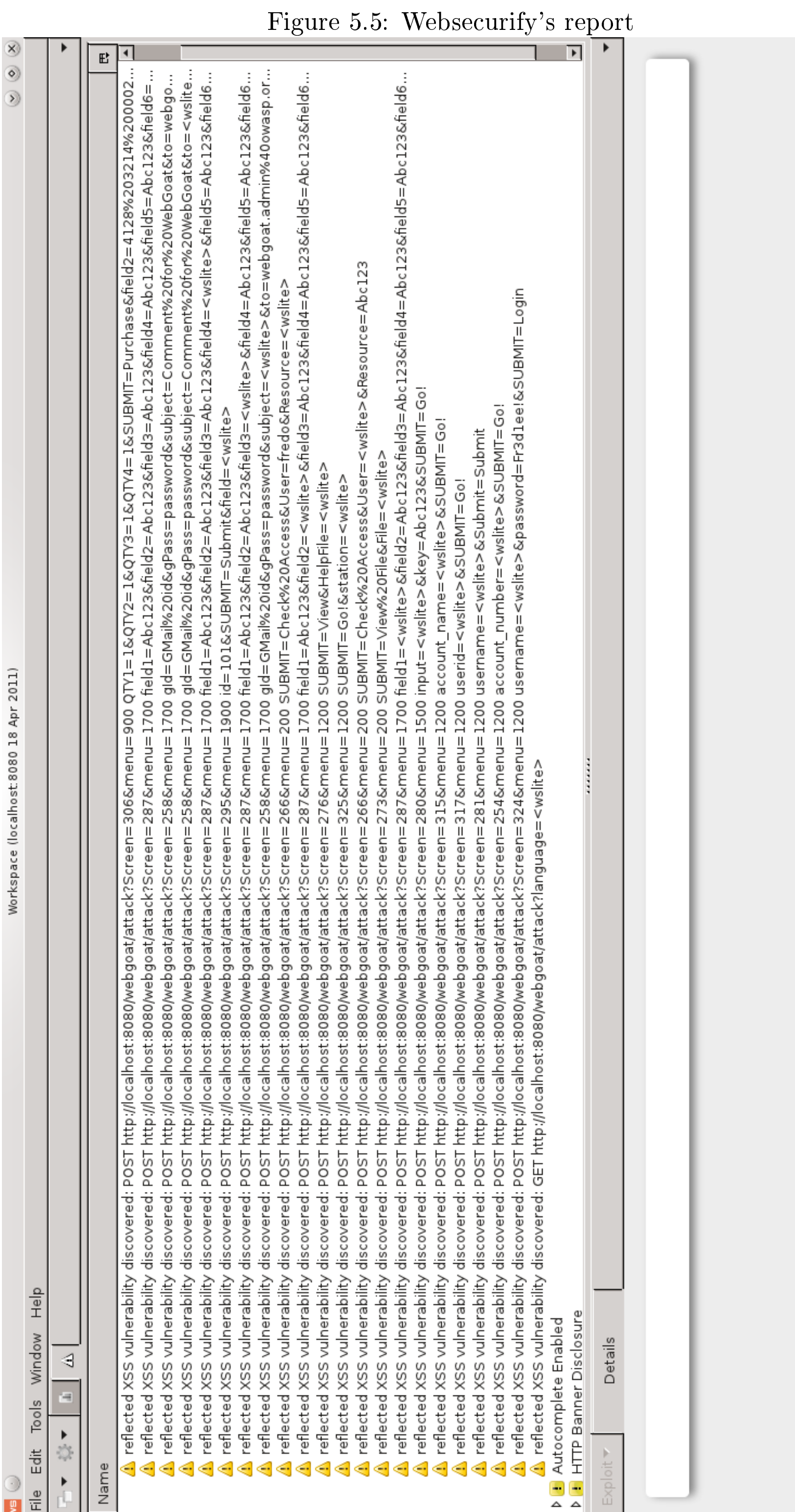


Figure 5.5: Websecurify's report

Table 5.2: Runtime against WebGoat in hh:mm:ss

w3af	01:13:11
Wapiti	02:04:16
Arachni	01:31:38
Websecurify	00:21:39

Table 5.3: Injection results of WebGoat, a question mark means that the value could not be determined, TP indicates the true positives and FP the false positives, FN indicates the false negatives

	SQL injection				Blind SQL injection				XPath injection			
	Total	TP	FP	FN	Total	TP	FP	FN	Total	TP	FP	FN
w3af	0	0	0	6	0	0	0	2	139	?	?	?
Wapiti	0	0	0	6	0	0	0	2	0	0	0	2
Arachni	2	2	0	4	0	0	0	2	0	0	0	2
Websecurify	0	0	0	6	0	0	0	2	0	0	0	2

Some of the tools also reported some other vulnerabilities that are not in the table. w3af reported three *unidentified vulnerabilities* using HTTP method GET, but testing these manually revealed no vulnerabilities. Arachni also reports a private IP address disclosure and several unencrypted password forms. Websecurify reports an HTTP banner disclosure and five email address disclosures.

5.2.3 phpBB

A larger test was performed on phpBB 2.0.0. Because it was an old version created for old PHP version the source code of phpBB had to be changed to get it to work. Running the tools on it proved to be a problem for most tools. The main problem is that every page of the forum uses a sid parameter as session id. The tools did not store this sid parameter, that is why the crawling phase took multiple hours. With such a high number of pages, the attack phase took multiple hours as well. Because of the amount of time the test had to be aborted manually. Only JSky was able to scan the entire site in a reasonable amount of time. The results can be found in table 5.6, 5.7, 5.8 and 5.9. Because the

Table 5.4: XSS, XST and CSRF results of WebGoat, numbers in brackets are the value when duplicates are included, a question mark means that it could not be determined, TP indicates the true positives and FP the false positives, FN indicates the false negatives

	XSS				XST				CSRF			
	Total	TP	FP	FN	Total	TP	FP	FN	Total	TP	FP	FN
w3af	3	1	2	6	1	?	?	?	0	0	0	3
Wapiti	87	7(17)	27(70)	0	0	0	0	1	0	0	0	3
Arachni	6	3	3	4	1	?	?	?	3	3	0	0
Websecurify	1	1	0	6	0	0	0	1	0	0	0	3

Table 5.5: Misc vulnerabilities results of WebGoat, numbers in brackets are the value when duplicates are included, TP indicates the true positives and FP the false positives, FN indicates the false negatives

	Path traversal				CRLF				Command injection			
	Total	TP	FP	FN	Total	TP	FP	FN	Total	TP	FP	FN
w3af	0	0	0	2	0	0	0	1	0	0	0	1
Wapiti	3	2(3)	0	0	0	0	0	1	0	0	0	1
Arachni	3	2	1	0	0	0	0	1	0	0	0	1
Websecurify	0	0	0	2	0	0	0	1	0	0	0	1

Table 5.6: Runtime against phpBB in hh:mm:ss

JSky	00:58:39
w3af	>05:00:00
Wapiti	>05:00:00
Arachni	>05:00:00
Websecurify	>05:00:00

total number of vulnerabilities is unknown, the number of false negatives in the tables is based on the highest number of true positives found by any tool for that category. For XST it could not be determined if the reported XST vulnerabilities are true or false positives, because the tools only test for XST by testing if the TRACE method is supported by the web server. The CRLF vulnerabilities could not be verified either. I was unable to reproduce it, but I was also unable to get the CRLF vulnerability in WebGoat to work.

5.2.4 Mutillidae

Another test was performed on Mutillidae[2]. Similar to WebGoat, it is also a set of vulnerable scripts, the author claims that the entire OWASP Top 10 is implemented. The tools were run with all modules and could try all links, except for three: the link to toggle hints, the link to toggle the security level and

[2] http://www.irongeek.com/i.php?page=security/mutillidae-deliberately-vulnerable-php-owasp-top-10

Table 5.7: Injection results of phpBB, TP indicates the true positives and FP the false positives,FN indicates the false negatives, this number could not be determined completely

	SQL injection				Blind SQL injection				XPath injection			
	Total	TP	FP	FN	Total	TP	FP	FN	Total	TP	FP	FN
JSky	0	0	0	≥ 0	0	0	0	≥ 0	0	0	0	≥ 0
w3af	0	0	0	≥ 0	0	0	0	≥ 0	0	0	0	≥ 0
Wapiti	0	0	0	≥ 0	0	0	0	≥ 0	0	0	0	≥ 0
Arachni	0	0	0	≥ 0	0	0	0	≥ 0	0	0	0	≥ 0
Websecurify	0	0	0	≥ 0	0	0	0	≥ 0	0	0	0	≥ 0

Table 5.8: XSS, XST and CSRF results of phpBB, a question mark means the value could not be determined, TP indicates the true positives and FP the false positives, FN indicates the false negatives, this number could not be determined completely

	XSS				XST				CSRF			
	Total	TP	FP	FN	Total	TP	FP	FN	Total	TP	FP	FN
JSky	3	2	1	≥ 0	1	?	?	?	0	0	0	≥ 0
w3af	0	0	0	≥ 2	11	?	?	?	0	0	0	≥ 0
Wapiti	0	0	0	≥ 2	0	0	0	≥ 0	0	0	0	≥ 0
Arachni	0	0	0	≥ 2	1	?	?	?	0	0	0	≥ 0
Websecurify	2	1	1	≥ 1	0	0	0	≥ 0	0	0	0	≥ 0

Table 5.9: Misc vulnerabilities results of phpBB, TP indicates the true positives and FP the false positives, FN indicates the false negatives, this number could not be determined completely

	Path traversal				CRLF				Command injection			
	Total	TP	FP	FN	Total	TP	FP	FN	Total	TP	FP	FN
JSky	0	0	0	≥ 0	2	?	?	?	0	0	0	≥ 1
w3af	0	0	0	≥ 0	0	0	0	≥ 0	1	1	0	≥ 0
Wapiti	0	0	0	≥ 0	0	0	0	≥ 0	0	0	0	≥ 1
Arachni	0	0	0	≥ 0	0	0	0	≥ 0	0	0	0	≥ 1
Websecurify	0	0	0	≥ 0	0	0	0	≥ 0	0	0	0	≥ 1

the link to reset the database. All the tools have an option to ignore certain pages or regular expressions of pages. In w3af this option can be found in the configuration menu for the "webspider" plugin. Wapiti and Arachni have a command-line option for this. The GUI of websecurify contains an input field where a regular expression of pages to ignore can be entered. However, to achieve this in Websecurify, the code had to be changed as it contains a known bug that causes URLs that should be excluded not to be excluded[3]. w3af failed to find the links with the vulnerable pages, therefore it was passed all pages as target and crawling was disabled. The results of this test can be found in table 5.10, 5.11, 5.12 and 5.13. Because every page of Mutillidae contains more than one vulnerability, the total number of false negatives could not be determined completely. Again, the number of false negatives is based on the highest number of true positives found by any tool for that type of vulnerability. The number of true and false positives for XST are unknown for the same reason as with the test against WebGoat: the tools only test if the webserver support the TRACE method. The reason the number of true and false positives for the CSRF vulnerabilities reported by w3af could not be determined is that w3af only reported that the vulnerabilities existed, but it did not specify where and what data it used to find these vulnerabilities.

A very peculiar result of this test is that the results of w3af are very poor. Testing w3af against certain pages of Mutillidae while a packet sniffer monitors the traffic and comparing this traffic with the traffic generated by other tools

[3]http://code.google.com/p/websecurify/issues/detail?id=53

Table 5.10: Runtime against Mutillidae in hh:mm:ss, w3af * is the result of the second test of w3af after changing Mutillidae

JSky	00:12:07
w3af	00:03:58
w3af *	00:07:50
Wapiti	00:03:34
Arachni	00:03:00
Websecurify	00:00:56

reveals the problem with w3af. It is the only tool that, in sending the HTTP POST request, does not send the name/value pair of the submit button, but only the name/value pairs of the other inputs of the HTML form (even though the name/value pair of the submit button is shown in its report). However, Mutillidae tests whether or not the form has been sent by checking for the name of the submit button in the POST data. The pages, therefore, do not "know" that w3af has entered data in the input fields and just show the original page with an empty form to fill in.

To get more results from w3af I have changed the check in the pages of Mutillidae. Instead of checking for the name of the button, the pages now check if the request method is POST. Running the test again with w3af produces the results of table 5.10, 5.11, 5.12 and 5.13 named "w3af *". Although w3af finds more vulnerabilities now, it still finds less than the other tools. A reason for this is a strange problem of w3af. When running w3af against a single page of Mutillidae in which it did not find any vulnerabilities during the test on all the pages of Mutillidae, it did detect the vulnerabilities. I have not been able to discover why these vulnerabilities where found when testing on a single page, but not when testing on all of the pages. A similar result appeared when w3af was two pages of Mutillidae were given as target to w3af: it only detected the vulnerabilities in one of the pages, however when given these pages as target separately it detected vulnerabilities in both of them.

Another problem of w3af that was discovered is that it does not fuzz textareas, only other inputs. Also, w3af appears to have some sort of cache. Changes in a page or in parameters of w3af only seem to take effect after restarting w3af.

Mutillidae also has a SQL injection in on of the pages that none of the tools could detect. This page (register.php) consists of five inputs: a field for the desired username, the password, confirmation of the password, signature and submit. All the tools only fuzzed one of the fields at a time. In this case that caused a problem. The password field was vulnerable for a SQL injection, however the script tested if the password field and the confirmation were the same. Because the tools only fuzzed one field at a time this was never the case, so the vulnerability in this field was not detected by any of the tools.

Also, two pages with an XSS vulnerability did not have any inputs. These contents of these pages were generated by the headers that were sent to retrieve the page. The vulnerabilities in these pages could be exploited by fuzzing the "Referrer" and "User-agent" header. However, none of the tools did this.

Table 5.11: Injection results of Mutillidae, w3af * is the result of the second test of w3af after changing Mutillidae, numbers in brackets are the value when duplicates are included, TP indicates the true positives and FP the false positives, FN indicates the false negatives, this number could not be determined completely

	SQL injection				Blind SQL injection				XPath injection			
	Total	TP	FP	FN	Total	TP	FP	FN	Total	TP	FP	FN
JSky	9	4(7)	2	≥ 5	0	0	0	≥ 8	0	0	0	≥ 0
w3af	0	0	0	≥ 9	0	0	0	≥ 8	0	0	0	≥ 0
w3af *	5	5	0	≥ 4	2	2	0	≥ 6	0	0	0	≥ 0
Wapiti	9	9	0	≥ 0	8	8	0	≥ 0	0	0	0	≥ 0
Arachni	9	9	0	≥ 0	0	0	0	≥ 8	0	0	0	≥ 0
Websecurify	9	9	0	≥ 0	0	0	0	≥ 8	0	0	0	≥ 0

Table 5.12: XSS, XST and CSRF results of Mutillidae, w3af * is the result of the second test of w3af after changing Mutillidae, numbers in brackets are the value when duplicates are included, TP indicates the true positives and FP the false positives, FN indicates the false negatives, this number could not be determined completely

	XSS				XST				CSRF			
	Total	TP	FP	FN	Total	TP	FP	FN	Total	TP	FP	FN
JSky	2	0	2	≥ 8	1	?	?	?	0	0	0	≥ 1
w3af	1	0	1	≥ 8	1	?	?	?	4	?	?	?
w3af *	10	4(9)	1	≥ 4	1	?	?	?	1	?	?	?
Wapiti	21	7(21)	0	≥ 1	0	0	0	≥ 0	0	0	0	≥ 1
Arachni	12	8	4	≥ 0	1	?	?	?	1	1	0	≥ 0
Websecurify	26	6	20	≥ 2	0	0	0	≥ 0	0	0	0	≥ 1

Table 5.13: Misc vulnerabilities results of Mutillidae, w3af * is the result of the second test of w3af after changing Mutillidae, numbers in brackets are the value when duplicates are included, TP indicates the true positives and FP the false positives, FN indicates the false negatives, this number could not be determined completely

	Path traversal				CRLF				Command injection			
	Total	TP	FP	FN	Total	TP	FP	FN	Total	TP	FP	FN
JSky	1	1	0	≥ 3	0	0	0	≥ 0	0	0	0	≥ 1
w3af	1	1	0	≥ 3	0	0	0	≥ 0	0	0	0	≥ 1
w3af *	3	3	0	≥ 1	0	0	0	≥ 0	1	1	0	≥ 0
Wapiti	16	4(16)	0	≥ 0	0	0	0	≥ 0	1	1	0	≥ 0
Arachni	3	3	0	≥ 1	0	0	0	≥ 0	1	1	0	≥ 0
Websecurify	0	0	0	≥ 4	0	0	0	≥ 0	0	0	0	≥ 1

Table 5.14: Injection results of zero.webappsecurity.com

	SQL injection	Blind SQL injection	XPath injection
HP WebInspect	43	0	0
w3af	12	1	10
Wapiti	2	1	0
Arachni	2	0	0
Websecurify	0	0	0

Table 5.15: XSS, XST and CSRF results of zero.webappsecurity.com

	XSS	XST	CSRF
HP WebInspect	48	0	2
w3af	6	1	8
Wapiti	7	0	0
Arachni	7	1	6
Websecurify	0	0	0

5.2.5 zero.webappsecurity.com

After installing the trial version of HP WebInspect it became clear that, next to a trial period of 15 days, a second limitation of the trial version is that only one particular web application could be scanned: http://zero.webappsecurity.com. Because this web application is used to sell HP WebInspect, it is likely that it is optimized for HP WebInspect. The results in tables 5.14, 5.15 and 5.16 seem to confirm this. Because of this reason and the fact that JSky complained about its license when trying to run it against this web application, the results of this test are not used, nor are the true and false positives counted.

Table 5.16: Misc vulnerabilities results of zero.webappsecurity.com

	Path traversal	CRLF	Command injection
HP WebInspect	0	0	0
w3af	0	0	0
Wapiti	1	0	0
Arachni	0	0	0
Websecurify	0	0	0

Chapter 6

Related work

6.1 General consensus

Comparisons between penetration testing tools have been performed by other people as well. In some of these comparisons the comparison was the goal ([13, 14, 15]) while in other papers these comparisons are used as a method to test a benchmark or a new technique for penetration testing tools ([5, 16, 17, 18, 19, 20]). The most common vulnerability that is tested for is SQL injection [13, 14, 15, 16, 17, 18, 19, 20] followed by XSS [14, 15, 5, 16, 17, 19]

Several of these papers conclude that the tools do not discover all vulnerabilities, [13, 14, 15, 5, 17, 18, 19], some also conclude that they produce a lot of false positives [13, 17, 18].

6.2 Description of each paper

For instance in [13] three commercial tools (HP WebInspect, IBM Rational AppScan and Acunetix Web Vulnerability Scanner) were used against 300 publicly available services to detect SQL injection, XPath injection, code execution, possible parameter based buffer overflow, possible username or password disclosure and possible server path disclosure. The names of the tools were not mentioned in the results to assure neutrality and because commercial licenses do not allow in general the publication of tool evaluation results, instead they were referred to as VS1.1, VS1.2 (two different versions of one tool), VS2 and VS3.

In [17] three commercials tools (the names of these tools are not given because of the same reason as in [13]) are tested against two web applications. One is custom made (MyReferences), the other one is the Online BooksStore. To test the tools several types of faults are injected in these web applications. Both web applications are tested for Cross Site Scripting and SQL injection for each type of fault injected. The conclusions are that different tools produce quite different results, all tools leave a considerable amount of vulnerabilities undetected and the tools result in a high amount (20% to 77%) of false positives.

Paper [18] is based on the results of [13]. The authors propose an approach to detect SQL injection. To test this approach the test in [13] is repeated. This time a tool created by the authors, referred to as VS.WS, that uses the proposed approach is tested as well. All the tools were tested against 262 public web

services and a Java implementation of four web services specified by the TPC-App benchmark. The only vulnerability that was tested for is SQL injection. They conclude that the results of their tool are better in both coverage and false positives than the commercial tools.

In [5] three tools (Burp Spider, w3af and Acunetix Web Vulnerability Scanner (Free Edition)) are tested on Django-basic-blog, Django-Forum and Satchmo online shop. Only reflected and stored XSS vulnerabilities are tested for, but the authors claim the same techniques also apply to other injection attacks such as SQL injection and directory traversal attacks. The conclusion is that coverage of the tools is low, however when the techniques from the authors are used (guided fuzzing and statefull fuzzing) more vulnerabilities are found.

In [16] a test suite for penetration testing tools is tested. The test is performed with four commercial and open source penetration testing tools for web applications, the names of these tools are not given in the paper. The tools are tested against an imitation of an online banking web application. This application contains several vulnerabilities, however the authors only test for XSS, SQL injection, blind SQL injection and file inclusion vulnerabilities. Each tool is used multiple times against the application, every time a different level of defense is used. Examples of such a level of defense are *typecasting* the input (casting the input to the type expected (e.g. integer or string)) or not showing MySQL errors. No conclusion is drawn on the quality of the tools, but the authors conclude that their test suite is effective for distinguishing tools based on their vulnerability detecting rates. They also conclude that no tool is able to find any vulnerability at level 2 or above.

In [19] penetration testing tools are tested to analyze the limitations of these tools. The author performs the test with the tools Grendel-Scan, Wapiti, w3af, Hailstorm, N-Stalker, Netsparker, Acunetix Web Vulnerability Scanner and Burp Scanner. The tools are tested against a modified version of the BuggyBank web application and, because the author deemed this not suitable for a full analysis of such tools because of its lack of features and functionality, against a secure and an insecure (with vulnerabilities intentionally implemented) version of the Hokie Exchange web application. The vulnerabilities that are implemented in the insecure version, and thus are tested for, are SQL injection, XSS, session management flaws, malicious file execution and buffer overflow. Of all of these vulnerabilities, except malicious file execution and buffer overflow, multiple types (e.g. reflected XSS, stored XSS etc.) are implemented. For example SQL injection from form inputs and from cookie variables. The conclusion of this thesis is that a testbed with secure and insecure versions of web applications is a suitable method to discover why penetration testing tools produce false positives and false negatives. The conclusion on the quality of the tools is that they perform relatively well in detecting the most simple kinds of SQL injection and XSS. However, to detect non-traditional instances of SQL injection and XSS, session management flaws, malicious file execution and buffer overflows, much work needs to be done to improve the techniques of the tools.

A similar conclusion is drawn in [14]. In this paper the penetration testing tools Acunetix Web Vulnerability Scanner, Cenzic Hailstorm Pro, HP WebInspect, IBM Rational AppScan, Mcafee SECURE, N-Stalker QA Edition, Qualys QualysGuard PCI and Rapid7 NeXpose are tested against older versions of Drupal, phpBB and Wordpress. The tools are also tested against a custom-built testbed with vulnerabilities that have a proven attack pattern. The vulnera-

bilities that were tested for are XSS (multiple types), SQL injection (multiple types), Cross-Channel Scripting, session management flaws, CSRF, information disclosure, Server and Cryptographic Configuration and Detection of Malware. The authors concluded that penetration testing tools are adept at detecting straightforward XSS and SQL injection vulnerabilities, but there is room for improvement for advanced and second-order forms of XSS and SQL injection, other forms of Cross-Channel Scripting, CSRF, and Malware Presence. The authors also note that the tools require a better understanding of active content and scripting languages, like SilverLight, Flash, Java Applets and Javascript.

In [20] a new tool CIVS-WS (Command Injection Vulnerability Scanner for Web Services) has been created based on a new approach by the authors to detected SQL/XPath injection. This tool is compared with HP WebInspect, IBM Rational AppScan, Acunetix Web Vulnerability Scanner and VS.BB (a tool that implements the approach proposed in [18]) by running them against operations implemented by nine web services. A part of these services is specified by the TPC-App benchmark another part is public code. Only SQL injection and XPath injection is tested for. The conclusion of the authors is that their tool (CIVS-WS) achieved a coverage of 100% and 0% false positives.

In [15] the tools Acunetix Web Vulnerability Scanner, IBM Rational AppScan, Burp Scanner, Grendel-Scan, Hailstorm, Milescan, N-Stalker, NTOSpider, Paros, w3af and HP WebInspect are tested against the web application WackoPicko that has been created by the authors of the paper. The vulnerabilities that were tested for are XSS (multiple types), SQL injection, command-line injection, file inclusion and file exposure. The conclusion of the paper is that crawling a modern web application is a serious challenge for penetration testing tools and that support for well-known, pervasive technologies (such as Flash and Javascript) should be improved, more sophisticated algorithms are needed to perform "deep" crawling and track the state of the application under test, and more research is warranted to automate the detection of application logic vulnerabilities.

6.3 Summary

Table 6.1: Summary of related work

Paper	Tools used
[13]	• Acunetix Web Vulnerability Scanner • HP WebInspect • IBM Rational AppScan
[14]	• Acunetix Web Vulnerability Scanner • Cenzic Hailstorm Pro • HP WebInspect • IBM Rational AppScan • McAfee SECURE • N-Stalker QA Edition • Qualys QualysGuard PCI • Rapid7 NeXpose
[15]	• Acunetix Web Vulnerability Scanner • Burp Scanner • Cenzic Hailstorm Pro • Grendel-Scan • HP WebInspect • IBM Rational AppScan • Milescan • N-Stalker QA Edition • NTOSpider • Paros • w3af
[5]	• Acunetix Web Vulnerability Scanner (Free Edition) • Burp Spider • w3af
[16]	Names are not given
[17]	Names are not given
[18]	• Acunetix Web Vulnerability Scanner • HP WebInspect • IBM Rational AppScan • VS.WS
[19]	• Acunetix Web Vulnerability Scanner • Burp Scanner • Cenzic Hailstorm Pro • Grendel-Scan • N-Stalker QA Edition • w3af • Wapiti
[20]	• Acunetix Web Vulnerability Scanner • HP WebInspect • IBM Rational AppScan • VS.BB • CIVS-WS

Chapter 7

Conclusion

The tools are able to find vulnerabilities in the test cases. However, there are certain vulnerabilities that none of the tools can find. Also, all tools produce false positives in certain test cases.

Section 7.1 is about the quaility of the tools divided into separate parts, in section 7.2 conclusions about the tools are drawn based on the results of the test cases. Section 7.3 contains the main conclusions of this thesis, section 7.4 is about the limitations of these tools, section 7.5 contains some suggestions for improvement, section 7.6 contains conclusions on every tool used in this thesis and section 7.7 is about how to choose a tool if one wants to test a web application.

7.1 Quality of the tools

The quality of the tools depend on three properties:

- quality of crawling
- quality of fuzzing
- quality of "analyzing"

7.1.1 Quality of the crawling

The crawler is responsible for crawling the web application for pages that may be vulnerable to a certain attack. The quality of the crawler can be determined by the number of pages it crawls. It should crawl all pages of the web application, but not include doubles. The crawling phase is independent of the type of weakness that is tested for or present in the web application.

Initially, all the tools failed to crawl the entire WebGoat, after changing the settings of the tools only the commercial JSky failed to crawl WebGoat.

For phpBB the opposite was true. Only JSky was able to crawl it like it should. The free tools managed to crawl it as well, but these crawled double pages with only a different sid as well. The sid consists of 32 hexadecimal digits, which means that the free tools will find every page 512 times. This causes the crawling phase to take a very long time.

Mutillidae was crawled successfully by all tools, except for w3af.

7.1.2 Quality of the fuzzing

The fuzzer is responsible for entering input that may expose a vulnerability. The quality of the fuzzer can be determined by what it inputs to find a certain vulnerability. These inputs should work for a certain vulnerability regardless of the underlying web- or SQL-server.

It is hard to tell how to determine the quality of the fuzzing as there exists no problem that is clearly caused by the fuzzing. However, some problems (like false negatives) may be caused by the fuzzing. By analyzing the cause of such a problem it can be determined if it was caused by the fuzzing phase.

7.1.3 Quality of the analyzing

The analyzer is responsible for analyzing the results of the inputs from the fuzzer. It should discover the vulnerabilities without producing false positives.

For WebGoat, only Arachni was able to find a few SQL injection vulnerabilities, the other tools did not because they could not be ran against WebGoat or did not recognize the error message from the web- or SQL-server.

Also, all the tools produced quite a lot of false positives when trying to detect XSS vulnerabilities in WebGoat.

For phpBB, none of the tools discovered all vulnerabilities. JSky and Websecurify discovered the most vulnerabilities, even though they both did not discover the command injection vulnerability. Only w3af discovered that vulnerability.

The results against Mutillidae are quite peculiar, because JSky performed less than the free tools. JSky is the only commercial tool in the test. One would expect that a program that costs money performs better than, or at the least equal to, free programs.

False positives are caused by the analyzing part, as it means it wrongly recognizes a vulnerability that is not there.

False negatives can be caused by the crawling, fuzzing or analyzing part. The crawling part may fail to reach the page with the vulnerability. The fuzzing part can cause false negatives by using "wrong" inputs that do not exploit a vulnerability. The analyzing part can be wrong in not recognizing a vulnerability that is there.

7.2 Conclusions per test case

The tools were tested by running them against several test cases. This section draws conclusions based on what every test case has shown about the use of these tools. Even though most of the test cases were not "real" web applications, the techniques they use are the same as in "real" web applications, therefore the results against the test cases are valid for use against "real" web applications as well. Some test cases revealed problems with a specific tool, these are explained in 7.5. Some test cases revealed problems with a specific tool, these are explained in 7.5.

7.2.1 General

- All test cases revealed that the tools only test for XST by testing if the web server supports HTTP TRACE. The tools do not check if this can be

abused

7.2.2 MiniSQLApp

- MiniSQLApp revealed that the tools are able to crawl a simple web application and can detect an SQL injection vulnerability; however, not all tools can detect blind SQL injection vulnerabilities

7.2.3 WebGoat

- WebGoat revealed that all the tested tools experience problems when authentication is used; however, for all the tools except JSky, could this be resolved
- WebGoat also revealed that the tools, except for Arachni, did not recognize the error message by the SQL server used by WebGoat
- WebGoat revealed as well that the tools test for XSS vulnerabilities by trying to inject an HTML tag, but the tools only test if this tag is present on the page, they do not test if this presence is on a location where the tag is parsed (i.e. not in an HTML attribute)

7.2.4 phpBB

- phpBB revealed that all the tools, except for JSky experience problems when a session id is used

7.2.5 Mutillidae

- Mutillidae revealed that none of the tools tries to perform an XSS attack by fuzzing the headers it sends to the web server
- Mutillidae also revealed that the tools only fuzz one input at a time, this caused the tools to miss an SQL injection in a form with two password fields that had to be equal

7.3 Main conclusions

The tools are able to find vulnerabilities in web applications and are, in general, easy to use, however as of yet they are by no means a complete solution to make a web application safe. The tools have quite a high number of false positives and false negatives, so a manual review of the false positives and the code in general (because of false negatives) is still necessary. The tools provide a good report of vulnerabilities that makes it pretty easy to verify the reported vulnerabilities (although in the test with Websecurify against WebGoat, Websecurify's report did not specify the pages with the discovered vulnerabilities correctly). However, because most of the reported vulnerabilities have to be verified manually (some of the reported vulnerabilities are clearly false or true positives and do not have to be tested), it can take hours (especially with a big web application) to verify all reported vulnerabilities to determine which are true positives and which are false. Determining the false negatives in a real web application is much harder as

Table 7.1: Final results of the tools against the different test cases, ++ = very good, + = good, +/– = average, – = bad, — = very bad, a question mark indicates that it could not be determined;[1]did not work initially, but could be solved

	MiniSQLApp			WebGoat			phpBB			Mutillidae		
	Crawling	Fuzzing	Analyzing	Crawling	Fuzzing	Analyzing	Crawling	Fuzzing	Analyzing	Crawling	Fuzzing	Analyzing
JSky	++	+/−	++	—	?	?	++	+	+/−	++	−	−−
w3af	++	++	++	+/−	−	−−	—	?	?	—	+/−	++
Wapiti	++	++	++	+/−	+/−	−−	—	?	?	++	+	−−
Arachni	++	++	++	+/−	+/−	+/−	—	?	?	++	+	+
Websecurify	++	+/−	++	+/−	−−	++	—	?	?	++	+/−	−−

they are not known. In the test web applications like WebGoat and Mutillidae it is possible, but very time consuming. To determine the false negatives and the cause of it the tools have to be run again against the specific page of the web application that produces the false negative, while monitoring what the tool sends to the web application and what it receives (e.g. with a packet sniffer). Determining false negatives in these cases will take several hours.

Table 7.1 contains the main conclusions on every tool per test case.

For crawling the deciding factor was the number of pages the tool could crawl. If it crawled all pages without requiring any configuration it gets the rating ++. If it fails to crawl all pages, but manages to crawl more than half of the number of pages it gets the rating +. If it only crawls about half of the number of pages or requires the user to help it gets the rating +/–. If it crawls less than half of the number of pages it gets the rating –. If it fails to crawl the web application completely it gets the rating —.

For fuzzing the deciding factor was the amount of true and false negatives. If the tool could find more than 90% of all vulnerabilities it get the rating ++. If it finds 60-89% it gets the rating +. If it only finds 40-59% it gets the rating +/–. If it finds 10-39% it gets the rating –. If it finds less than 10% it gets the rating —.

For analyzing the deciding factor was the amount of false positives and duplicates. If the tool produced less than 10% false positives and duplicates it gets the rating ++. If the tools produced 11-39% false positives and duplicates it gets the rating +. If it produced 40-59% false positives and duplicates it gets the rating +/–. If it produces 60-89% false positives and duplicates it gets the rating –. If the tool produced more than 90% false positives and duplicates it gets the rating —.

7.4 Limitations

All of the tools seem to have problems when web applications use techniques that are a bit more advanced than average pages, for instance cookies for logging in and session ids. When these tools are used, they should be monitored carefully to detect and solve these problems.

7.5 Suggestions for improvement

In this section suggestions to improve the tools will be given. This will contain both suggestions for the tools in general and specific suggestions for each tool.

7.5.1 Suggestions in general

Improving the crawling

- the tools should be able to handle session ids in the URL better, this can either be done by allowing the user to specify this id (like with cookies) or the tools should test if the only difference in pages they are crawling is the session id; this suggestion is based on the results of the test against phpBB

- right now the tools only fuzz one input at a time, in certain cases a vulnerability can be found by fuzzing multiple inputs at the same time; this problem was revealed by Mutillidae

Improving the fuzzing

- the tools should also try to fuzz headers (like the referrer or the user-agent) as well as "normal" inputs; this suggestion is based on the results of the test against Mutillidae

- currently the tools only test for XST vulnerabilities by checking if the web server supports HTTP TRACE, the tools should also try to abuse this functionality; this suggestion is based on the results of all test cases

Improving the analyzing

- when testing for XSS vulnerabilities, the tools should check where the HTML code appears to see if the code would be executed/parsed by the browser (i.e. the code does not appear in a textarea or a similar field); this suggestion is based on the results of the test against WebGoat

- when testing for XSS vulnerabilities, the tools should precede the HTML code they inject with "> to close any tag in which the code may appear (e.g. an input field or link); Wapiti already does this in certain cases, presumably only in GET parameters; right now the tools only test if the injected text is present on the page, however this might be inside an HTML tag, for instance <input type="text" value="*injected text*">; if this is the case, the injected text is not parsed by the browser; if the tools precedes the injected text with "> the resulting HTML will be <input type="text" value="">*injected text*">; this way the browser will parse the injected text; ; this suggestion is based on the results of the test against Mutillidae

- the list of error messages that shows SQL injection is possible should be increased; this suggestion is based on the results of the test against WebGoat

7.5.2 w3af

Improving the crawling

- use textareas for fuzzing as well, just like the other tools, instead of only inputs that are created by the input HTML tag; this suggestion is based on the results of the test against Mutillidae
- handle multiple pages better, so that all vulnerabilities can be found regardless of the number of pages that is set as target; this suggestion is based on the results of the test against Mutillidae

Improving the fuzzing

- send the name/value pair of the submit button in the HTTP POST data to the page as well as the other inputs; this suggestion is based on the results of the test against Mutillidae

7.5.3 Arachni

Improving the fuzzing

- change the test for SQL injection, right now it adds *'-;'* to the default value or *1'-;'* if there is no default value, this does not result in a SQL injection in all servers; a better input would be an odd number of quotes (single and double); this suggestion is based on the results of the test against WebGoat

7.6 Conclusion about each tool

JSky was a bit of a disappointment as well, especially as it is a commercial tool. Despite this fact it failed against WebGoat and was outperformed by the free tools against the other test cases, with the exception of XSS vulnerabilities in phpBB.

w3af scored mediocre. It did discover a number of vulnerabilities, but it never excelled in any category or test case. Except for command injection in phpBB, other tools discovered more vulnerabilities in the other categories and test cases.

Except for the phpBB and zero.webappsecurity.com case studies, Wapiti discovers a lot of vulnerabilities. However, a lot of these are double or false positives. If only unique true positives are taken into account, it still discovers a high number of vulnerabilities, although it failed to detect any vulnerability in phpBB.

Arachni performs pretty good as well. It is the only tools that discovered any SQL injection vulnerabilities in WebGoat. For the other tests it performed quite similar to Wapiti, but with less doubles and false positives.

Websecurify was a bit of a disappointment. In all test cases it was by far the fastest program and because of its GUI the easiest to use, however despite its claim that it detects the entire OWASP top 10, in the test cases it only detected XSS and SQL injection vulnerabilities. Also, it was the only tool in the test that failed to detect blind SQL injection.

7.6.1 General

It is impossible to name a tool that is the best. The reason is that the usefulness of the tools depends on what web application is going to be tested and what vulnerabilities is going to be tested for.

As mentioned by several papers, [13, 14, 15, 5, 17, 18, 19], and the false negatives in the tests none of the tools are able to discover all vulnerabilities. Also, the tools have problems when the web application that is being tested is not a straightforward static web page. When the web application uses techniques like authentication or session ids the tools have problems with crawling the pages of the web applications that use these techniques.

Other papers mention that the tools produce a lot of false positives [13, 17, 18], in the tests in this thesis it appeared that the tools do indeed produce quite a high number of false positives. The tests in this thesis also showed that some tools produced duplicates and had problems with crawling certain web applications. None of the related work mention any of these problems.

7.7 Choosing a tool

When a session id is used the only tool in this test that can handle it correctly is JSky. However, if HTTP authentication is used with a cookie, none of the tools works by default. However, all tools except JSky can be made to work in this case. In these tests, the most vulnerabilities are found by Wapiti and Arachni, although Wapiti produces a lot of duplicate results and quite a high number of false positives. However, these results only apply to the test cases used in this thesis, for other test cases the results may differ. Also, as none of the tools is able to find all vulnerabilities in a web application, currently the best way to find as many vulnerabilities as possible is to use multiple tools.

Chapter 8

Future work

Obvious ideas for future work would be to test other tools and vulnerabilities. However, because of the amount of tools, vulnerabilities and test cases this can go on for a very long time. A suitable way to test such tools is to divide it into two separate parts:

1. Crawling
2. Fuzzing and Analyzing

The quality of crawling is independent of the quality of fuzzing and analyzing. These two parts can both be tested separately. To test the fuzzing and analyzing web applications like WebGoat and Mutillidae can be used, they use the same techniques as "real" web applications with the advantage that the vulnerabilities that are present are known. To test the crawling one would have to find or create a test suite that uses several techniques to test the crawling part.

Bibliography

[1] Howard, M., LeBlanc, D., Viega, J.: 24 Deadly Sins of Software Security: Programming Flaws and How to Fix Them. 1 edn. McGraw-Hill, Inc., New York, NY, USA (2010)

[2] Oehlert, P.: Violating Assumptions with Fuzzing. IEEE Security and Privacy **3** (2005) 58–62

[3] Palmer, S.: Web Application Vulnerabilities: Detect, Exploit, Prevent. Syngress Publishing (2007)

[4] Sutton, M., Greene, A., Amini, P.: Fuzzing: Brute Force Vulnerability Discovery. Addison-Wesley Professional (2007)

[5] Mcallister, S., Kirda, E., Kruegel, C.: Leveraging User Interactions for In-Depth Testing of Web Applications. In: Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection. RAID '08, Berlin, Heidelberg, Springer-Verlag (2008) 191–210

[6] Zhao, G., Zheng, W., Zhao, J., Chen, H.: An Heuristic Method for Web-Service Program Security Testing. ChinaGrid, Annual Conference **0** (2009) 139–144

[7] HP: HP WebInspect (2009) http://www.hp.com/spidynamics/products/webinspect/.

[8] NOSEC: JSky (2010) http://www.nosec-inc.com/en/products/jsky/.

[9] Riancho, A.: w3af (2011) http://w3af.sourceforge.net.

[10] Surribas, N.: Wapiti (2009) http://wapiti.sourceforge.net.

[11] Laskos, A.: Arachni - Web Application Vulnerability Scanning Framework (2011) https://github.com/Zapotek/arachni.

[12] : Websecurify (2011) http://www.websecurify.com.

[13] Vieira, M., Antunes, N., Madeira, H.: Using Web Security Scanners to Detect Vulnerabilities in Web Services. 2009 IEEEIFIP International Conference on Dependable Systems Networks (2009) 566–571

[14] Bau, J., Bursztein, E., Gupta, D., Mitchell, J.: State of the Art: Automated Black-Box Web Application Vulnerability Testing. In: Proceedings of the 2010 IEEE Symposium on Security and Privacy. SP '10, Washington, DC, USA, IEEE Computer Society (2010) 332–345

[15] Doupé, A., Cova, M., Vigna, G.: Why Johnny can't pentest: an analysis of black-box web vulnerability scanners. In: Proceedings of the 7th international conference on Detection of intrusions and malware, and vulnerability assessment. DIMVA'10, Berlin, Heidelberg, Springer-Verlag (2010) 111–131

[16] Fong, E., Gaucher, R., Okun, V., Black, P.E., Dalci, E.: Building a Test Suite for Web Application Scanners. In: Proceedings of the Proceedings of the 41st Annual Hawaii International Conference on System Sciences. HICSS '08, Washington, DC, USA, IEEE Computer Society (2008) 478–

[17] Fonseca, J., Vieira, M., Madeira, H.: Testing and Comparing Web Vulnerability Scanning Tools for SQL Injection and XSS Attacks. Pacific Rim International Symposium on Dependable Computing, IEEE **0** (2007) 365–372

[18] Antunes, N., Vieira, M.: detecting SQL Injection Vulnerabilities in Web Services. Dependable Computing, Latin-American Symposium on **0** (2009) 17–24

[19] Shelly, D.A.: Using a Web Server Test Bed to Analyze the Limitations of Web Application Vulnerability Scanners. Master's thesis, Virginia Polytechnic Institute and State University, Blacksburg, Virginia (July 2010)

[20] Antunes, N., Laranjeiro, N., Vieira, M., Madeira, H.: Effective Detection of SQL/XPath Injection Vulnerabilities in Web Services. In: Proceedings of the 2009 IEEE International Conference on Services Computing. SCC '09, Washington, DC, USA, IEEE Computer Society (2009) 260–267

Appendix A

MiniSQLApp

index.php

```
<html>
<head>
<title>Test SQL injection</title>
</head>
<body>
<a href="index1.php">other page</a>
<a href="index2.php">another page</a>
</body>
</html>
```

index1.php

```
<html>
<head>
<title>Test SQL injection</title>
</head>
<body>
<?php
if ($_SERVER['REQUEST_METHOD'] == "POST")
{
        mysql_connect("localhost", "root", "");
        mysql_select_db("test");
        $query = "SELECT * FROM test WHERE name='" . $_POST['name'] . "'";
        $result = mysql_query($query) or die (mysql_error());
        if (mysql_num_rows($result) == 0)
        {
                echo "Not found";
        }
        while ($row = mysql_fetch_assoc($result))
        {
                echo $row['id'] . " => " . $row['name'] . "<br>\n";
        }
}
?>
```

```
<form name="form" action="" method="post">
<input type="text" name="name"> Name<br>
<input type="submit" name="smb" value="Find name">
</form>
</body>
</html>
```

index2.php

```
<html>
<head>
<title>Test SQL injection</title>
</head>
<body>
<?php
if ($_SERVER['REQUEST_METHOD'] == "POST")
{
        mysql_connect("localhost", "root", "");
        mysql_select_db("test");
        $query = "SELECT * FROM test WHERE name='" . $_POST['name'] . "'";
        $result = mysql_query($query);
        if (mysql_num_rows($result) == 0)
        {
                echo "Not found";
        }
        while ($row = mysql_fetch_assoc($result))
        {
                echo $row['id'] . " => " . $row['name'] . "<br>\n";
        }
}
?>
<form name="form" action="" method="post">
<input type="text" name="name"> Naam<br>
<input type="submit" name="smb" value="Find name">
</form>
</body>
</html>
```