ZAVOD ZA ELEKTRONIKU, MIKROELEKTRONIKU, RAČUNALNE I INTELIGENTNE SUSTAVE
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA
SVEUČILIŠTE U ZAGREBU

# EXPLOITING AND AUTOMATED DETECTION OF VULNERABILITIES IN WEB APPLICATIONS

Vanja Suhina

Zagreb, 2007.

# Contents

# 1. Introduction

Web applications have become common way for companies to conduct business with the outside world. The static web pages are gone and now, almost every company has its own dynamic, interactive Web application that communicates with their clients. Often, these applications are written by programmers not familiarized with security problems behind their Web application front-end or their work is guided by hard to get deadlines so application security is not very high on their priority list. Thus these Web applications become main threat to secrecy and integrity of company's sensitive data.

The Open Web Application Security Project (OWASP) [1.] is a non-profit organization that provides unbiased, practical, cost-effective information about application security. The OWASP community develop tools, teaching environments, guidelines and other material to help organizations and individuals learn about application security. All projects are licensed under open source license. In July 2007, they released revision of the OWASP Top 10 document [2.] that dated back to 2004 and covered most popular Web application vulnerabilities and attacks at that time. The new OWASP Top 10 2007 is based on MITRE's Vulnerability trends for year 2006 [3.] and it describes most common Web application security issues. This list will be used in this paper to enumerate vulnerabilities and explain how they can be exploited to attack web sites.

Few open source tools that check for vulnerabilities in web applications will be examined. If it is possible to automatically scan for vulnerability, it will be shown how the tools do it. All the tools that we will examine will be black-box scanners, that is, tools that examine application from the outside and doesn't have information on application's source code or logic. Static code analysis tools won't be used in this paper.

The paper is structured as follows. In the second section, overview of OWASP Top 10 list is given. For each vulnerability, the description with an example is given and then it is described how the attacker can exploit this vulnerability. If there are various types of vulnerability in one list item, subsections describes them separately. If there are some popular real life examples of exploiting the certain vulnerability, they are also given. In third section, it is shown what are the methods to perform automated scanning, and how various tools do it. If scanners aren't capable of searching for that kind of vulnerability, it is explained why. The paper finishes with conclusion in section 4, and list of references in section 5.

# 2. Exploiting vulnerabilities

The purpose of this section is to describe most common vulnerabilities and their exploitation. The list of most common vulnerabilities is taken from the OWASP Top 10 2007 document and it covers the following ones:

A1. Cross Site Scripting (XSS)

A2. Injection Flaws

A3. Malicious File Execution

A4. Insecure Direct Object Reference

A5. Cross Site Request Forgery (CSRF)

A6.Information Leakage and Improper Error Handing

A7. Broken Authentication and Session Management

A8. Insecure Cryptographic Storage

A9. Insecure Communications

A10. Failure to Restrict URL Access

Some of the vulnerabilities can be split into different sub-categories, so then, each category and description of exploitation will be given. In some cases, there were interesting real life examples of exploiting particular vulnerability (or several of them) and then those will also be mentioned.

## 2.1. Cross Site Scripting (XSS)

"XSS is the New Buffer Overflow", says Jeremiah Grossman, one of the authors of the book "Cross Site Scripting Attacks: XSS Exploits and Defense". Indeed, what were buffer overflows to stand alone applications, XSS is to the Web applications. Cross Site Scripting is becoming more and more popular, and as much as 80% of Web applications are vulnerable to XSS [4.].

XSS is vulnerability in which malicious data is embedded into HTML code during the dynamic creation of the page. The malicious data can be embedded either on the server side when Web application constructs web page to respond to the request or on the client side when Document Object Model (DOM) is being built and client side scripts are being executed.

In both cases, the problem is that data that is used to create the page and that can be controlled by an attacker is not being properly encoded or filtered. Therefore, an attacker can inject his own code and execute it via supported scripting language, most often Javascript.

There are three types of XSS vulnerabilities and they differ on when and how the malicious data is injected into web page:

1. Non-persistent or Reflected XSS
2. Persistent or Stored XSS
3. DOM Based or Local XSS

### 2.1.1. Non-persistent or Reflected XSS

This is most common and simple XSS scenario. An attacker crafts link with malicious data that points to the vulnerable web site and tricks the victim to visit it. After victim opens link, vulnerable application takes malicious data from the parameters, embeds it in HTML page and echoes it back to the user. The link can look like this:

```
http://www.somesite.com/search.php?query=<script>alert('xss')</script>
```

On the web server, in `search.php`, there is vulnerable code that can look like this:

```
<? echo("<p>Search results for query: ".$_GET['query'].".</p>"); ?>
```

As it can be seen above, the parameter '`query`' is echoed back to the user without encoding special characters. This results in the following HTML code in the requested page.

```
<p>Search results for query: <script>alert('xss')</script>.</p>
```

The script was echoed back to the victim and it executes in victim's browser.

When exploiting reflected XSS, the attacker need to trick the user to follow the malicious link. The goal of the attack can be stealing personal and sensitive data, session hijacking or something else. For example, if the attacker finds XSS hole in some bank site that the victim uses, she/he can trick the victim to follow the link to the bank's web site with the malicious payload in the parameter. The malicious payload can be used for stealing the cookie:

```
<script>var img=new Image();img.src='http://www.attacker.com/log.php?cookie='
 +document.cookie</script>
```

If the victim is logged in to the bank site or the credentials are sent automatically when accessing the web site, the victim's credentials stored in the cookie, will be sent via a parameter to the attacker's web site. The attacker can then use this cookie to impersonate victim and access his/hers private data.

### 2.1.2. Persistent or Stored XSS

The goal of attack in Stored XSS is the same as in Reflected, but the difference is in where and when the malicious payload is echoed back to the user. The payload does not have to be echoed back immediately. It is stored on the server in database, file system or some other location and it is retrieved from there during creation of some page and echoed back to any user that requested it. It is possible that it's echoed back immediately, but the main point is that the server embeds payload to the page every time it is requested, and it is no more necessary to trick the user to follow a malicious link. This automatically targets more users. So, if an attacker injects malicious payload through some vulnerable page that can look like this:

```
http://www.someblog.com/postcomment.php?comment=<script>alert('xss')</script>
```

the payload can be echoed to all users through some public page:

```
http://www.someblog.com/publiccomments.php
```

or it can target some particular user or administrator:

```
http://www.someblog.com/userX/mycomments.php
http://www.someblog.com/admin/allcomments.php
```

The payload that is used in this attack can be similar to those payloads used in non-persistent XSS attacks. Storing the payload on the server and echoing it back to all the users makes the attack widespread and therefore, it is the best scenario for attackers. The attackers can use this vulnerability for stealing user's personal data, session hijacking, web defacements or performing phishing scams. There are several examples of XSS worms that used this type of XSS and they are going to be mentioned in chapter 2.1.4.

## 2.1.3. DOM Based or Local XSS

Unlike the first two, this type of XSS vulnerability doesn't rely on embedding the data on the server side. The web browser on the client side is responsible for connecting Javascript functions with malicious data injected in page URL, HTTP headers or somewhere else. Let's assume there is a fragment of HTML code like this:

```
<p>You came here from: <script>document.write(document.referrer)</script>/p>
```

This peace of code is intended to write on screen from what page the user comes. The `document.referrer` is Document Object Model (DOM) Component that is populated during the construction of the page from the browser's point of view. It actually looks in the HTTP response headers and copies the value of header: `Referer`. If the attacker can force user to visit this page from page that has a malicious payload in URL, the malicious payload will first be stored in HTTP header `Referer`, and then, in the victim's browser, it will be embedded to HTML page and executed. The URL with malicious payload could look like this:

```
http://www.maliciouswebsite.com/somepage.html?<script>alert('xss')</script>
```

If the victim visits the malicious web site and then goes to vulnerable page from mentioned before, the following will be embedded to HTML.

```
<p>You came here from:
 http://www.maliciouswebsite.com/somepage.html?<script>alert('xss')</script></p>
```

This will execute the script and it is a point where XSS occurs. Some other DOM objects that can be manipulated and exploited are: `document.URL`, `document.URLUnencoded`, `document.location`, `window.location`, etc. The main problem here is in web browsers that doesn't encode special characters. At the time of writing, Internet Explorer 7 still doesn't encode characters '<' and '>', while Firefox and Opera does. This makes Internet Explorer users vulnerable to this attack but other browsers may also be vulnerable to other attack vectors that don't use these two characters.

When trying to attack DOM Based XSS, attacker can do most harm if she/he finds a hole on some file on local system. An attacker can trick victim to execute Javascript code that will open vulnerable page on local system with malicious data appended to the page. When a local file is opened and malicious code is being executed, it executes outside the browser’s sandbox, under logged in user’s privileges and it leads to remote code execution.

### 2.1.4. Real-life examples of exploiting XSS

This chapter will take a look at some examples of exploiting XSS vulnerability.

It was shown that it is possible to write XSS worms which can self propagate through the net and infect various users on the domain.

The first major and most famous worm of this kind was The Samy Worm [7.] which struck MySpace.com in October 2005. It used a hole in the filtering mechanism that should have blocked all the malicious content if a user tried to enter one. The author of the worm, Samy, succeeded in avoiding filter rules and managed to upload the malicious code to his own profile. The malicious code was embedded in his profile and served to every user that visited it, thus making it a typical example of exploiting stored XSS vulnerability. The malicious code did not only infect the users that opened his profile, but also used the hole in MySpace to copy the source code to the visitor's profile, so it began self-propagating. Over the period of 24 hours, it infected over 1.000.000 users before MySpace had to shutdown his servers. It set a record in fastest propagation leaving far behind worms such as Blaster, Slammer or Code Red. The good thing in this story is that “malicious” code had not done anything malicious. It added Samy to the victim's friends list and added to the profile a message: “but most of all, Samy is my hero.”

There is a proof-of-concept XSS worm written by security researcher Rosario Valotta who managed to wrote first worm that propagates across four different domains, all web mail providers popular in Italy [8.]. Nduja Connection worm targets Libero.it, Tiscali.it, Lycos.it and Excite.com and exploit XSS vulnerabilities found in each one of them that allow injection of malicious code within body of an e-mail. Therefore, the code executes if the victim just opens the malicious e-mail. The worm collects e-mails from the Inbox, extract contacts e-mail addresses and self propagates to those contacts, but as it is just a proof of concept, it doesn't have any malicious purpose.

In July 2007, a German researcher Banjamin Flesch wrote first friendly XSS Worm that propagates through Wordpress blogs and tries to fix the vulnerability issues [9.]. It uses AJAX, and exploits XSS and CSRF vulnerabilities.

## 2.2. Injection Flaws

Injection flaws are second most often flaws in Web applications. Only SQL injections cover 13% of all vulnerabilities reported in year 2006 [4.] and every forth site is vulnerable to SQL injection. Other injection flaws such as LDAP, XPath, etc. adds to this numbers which makes injection flaws very widespread and exploitable.

Injection flaws happen when user’s data is used in constructing queries or statements to the underlying system without filtering. Malicious input can change content and logic of query or statement and interpreter can execute attacker’s commands. Most often is SQL Injection, but there are others such as LDAP, XPath, XML, XSLT, OS, etc. In this section, it will be given the description of the following injection flaws:

1. SQL Injection
2. XPath Injection
3. LDAP Injection
4. CRLF Injections
5. OS Commanding
6. SSI Injection

7. MX Injection

There are also some other types of injection flaws but they are not covered here.

## 2.2.1. SQL Injection

SQL Injection occurs when user's data is used to construct a SQL statement but the data is not properly filtered or checked for type. This can lead to changing the meaning of the query and malicious actions can take place. For example, let's look at the query that is constructed in this way:

```
query="SELECT data FROM users WHERE name='" +$name + "' AND pass='" +$pass +"';"
```

If variables `name` and `pass` are not checked for special characters, the attacker can change the meaning of the query by using the following input for the `pass` variable:

```
a' OR '1'='1
```

This changes the query condition to always return true, and so, it bypasses the authentication for particular user. If the attacker doesn't have any username, he/she can use the similar trick to log in as a first user in a database table.

Escaping the quotes isn't enough, because the attacker can launch some attacks without quote characters or even without any special characters. If the query is structured as follows:

```
query = "SELECT data FROM users WHERE userid=" + $userid
```

the attacker can use the following value for the `userid` parameter:

```
123 or userid is no null
```

In this case, the absence of a data type check is considered a vulnerability.

Attacking injection flaws have different goals then XSS. The intention behind injecting your own code into underneath queries is to change logic of the query and access information that is stored in database and that shouldn't be accessible to the user. The goal is to steal sensitive data or other user's credentials. It can also be used to bypass authorization mechanism.

In all types of injection flaws, an attacker changes data that is passed to interpreter and tries to find out if some security mechanisms are enforced. Very often, the query won't be correct and if error handling is not well configured, the message with information about error in query and underlying system is exposed to attacker. The attacker's goal can be to form valid query which will return not only intended, but also sensitive data to the client.

If error handling is done well and Web application doesn't disclose any information when query is invalid, other approach can be used. If injection still happens, but nothing echoes back to the client, the query logic can be changed so that attacker adds another condition to the query that can be either true or false. When condition is true, the valid page should appear, and when condition is false,a non descriptive error page is displayed. However, it is now possible to perform new queries to the database that will return true or false and observe the results. This technique is called Blind SQL Injection [13.] and it can be used to harvest entire database.

If the underlying database is MS SQL it may be possible to run code on the remote server using the SQL injection. There is a system procedure in MS SQL called xp_cmdshell [21.] that can be run if user that is connected to the database has appropriate rights. It is used to issue system commands to the

Windows command shell. The attacker can perform such attack that passes his commands to this procedure and then echoes back the system call results. It appears then that the user is logged in to the remote machine and can use it's shell.

### 2.2.2. XPath Injection

If a database is substituted by an XML document, SQL queries are substituted by XPath queries. If there is no proper validation of user's data used in queries, XPath injection occurs [14.]. The principle is the same as in SQL injection attacks. The attacker's data is not filtered and it is used to construct XPath queries. The attacker's injected code changes the meaning of the query and access private information If the user search an XML document for user named xxx whose password is yyy the query may look something like this:

```
/users/user[username = 'xxx' and password = 'yyy']
```

Changing the value of the password to some value that adds always true condition to the query, it is possible to bypass authentication. The query may now look like this:

```
/users/user[username = 'xxx' and password = 'yyy' or '1'='1']
```

Beside authentication bypass, XPath injection vulnerability can be used to retrieve the whole backend XML document. This is done through series of XPath queries and it is called Blind XPath Injection.

However, under some circumstances, it is possible to retrieve the whole XML document with single query. The attack is called XPath SQUAT and stands for XPath Single Query Attack. If AJAX application performs XPath query on the server but the data is sent to the client in raw XML format and there it is transformed to HTML, the following query will retrieve the whole document.

```
'] | /* | /foo[bar='
```

The query first closes the current condition ( `']` ), then it uses the join operator ( `|` ) with condition that selects all the nodes starting from the root ( `/*` ) and finally it joins another condition that makes sure that query is not broken because of closing the first condition.

As it was shown, XPath Injection can lead to bypassing the authentication protocol and stealing whole documents that contains application's data.

### 2.2.3. LDAP Injection

Lightweight Directory Access Protocol (LDAP) is a widely used protocol for accessing information directories. If Web application uses client's data to construct LDAP queries, and it is not filtering and validating data, LDAP Injection can occur [15.]. For example, web application parameter can be transferred directly to LDAP server without filtering. Query in web browser can look like this:

```
http://www.someapplication.com/ldap-queryuser.asp?name=xxx
```

If the value of the `name` parameter is directly transferred to LDAP query, it is possible to inject more code, change the meaning of the query and access information that the user shouldn't be able to access. For example, the following two queries may return the user's password and enumerate all the users:

```
http://www.someapplication.com/ldap-queryuser.asp?name=xxx)(|password=*)
http://www.someapplication.com/ldap-queryuser.asp?name=*
```

If a Web application is vulnerable to LDAP injection, it can give the attacker much info about the system, users and their private information.

### 2.2.4. CRLF Injection

CRLF Injection is attack when two special characters (Carriage Return and Line Feed) are injected into HTTP header values thus making possible to inject new HTTP header in response. This can be done only if there is no validation of user's data.

Some Web applications have a need to include users data to response headers. Breaking out of a specified header and making it possible to add another ones is a risk. The attacker can use this vulnerability to perform HTTP Response splitting attack [16.].

In this attack, the attacker injects his code into HTTP response headers. The code interrupts the current HTTP response and crafts another one splitting TCP stream in two parts. When the victim requests another page it will receive attacker crafted malicious HTTP response. For example, let's say that Web application takes `language` parameter from the user and embeds it in the `Location` HTTP response header. This can be used to redirect the user to the appropriate page regarding the language he chose. If the attacker inserts his own payload into the `language` parameter, the malicious payload is passed to HTTP response header. The attacker can use this code for example:

```
foobar%0d%0aContent-Length:%200%0d%0a%0d%0a
HTTP/1.1%20200%20OK%0d%0a
Content-Type:%20text/html%0d%0a
Content-Length:%2027%0d%0a%0d%0a<html>Attackers HTML</html>
```

This results in the following TCP stream:

```
HTTP/1.1 302 Moved Temporarily
Date: Wed, 24 Dec 2003 15:26:41 GMT
Location: http://somesite/lang.jsp?lang=foobar
Content-Length: 0

HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 27

<html>Attackers HTML</html>
Content-Type: text/html
... the rest of the original HTTP response headers and body
```

Second HTTP response is matched to the second request the user makes and it is supposed to fool the victim. Using the HTTP Response splitting attack the attacker can perform various new attacks such as cross user defacement, web cache poisoning, hijacking pages with sensitive user information or XSS attacks.

### 2.2.5. OS Commanding

OS Commanding [18.] is a vulnerability that allows the Web application to pass user commands to the underlying operating system and execute them. This is due to improper input validation. Web application can use user's data to open file the user has specified:

```
http://www.someapplication.com/cgi-bin/info.pl?name=John&template=tmp1.txt
```

The attacker can change the value of `template` parameter to:

```
/bin/ls|
```

which will list all the files and folder and pipe the output to the open statement that was used to open the template file.

If the application uses exec statement with some user provided parameters, it is possible to force the Web application to execute another command by appending a semicolon. Let's say that exec statement in the code looks like this:

```
exec("ls -la $dir",$lines,$rc);
```

Then, the attacker can use this value for the `dir` parameter:

```
%3Bcat%20/etc/passwd
```

With the ability to execute commands on the server, the attacker can use this vulnerability to upload malicious programs to the server and mount some other attacks.

### 2.2.6. SSI Injection

Server Side Include (SSI) [20.] is a simple server-side scripting language that is most often used for including content of one file into another or running OS commands on the server. The basic syntax that makes the server do so is [19.]:

```
<!-- #directive parameter=value parameter=value -->
```

The most common directives that are used are `include`, `exec` and `echo` and here are few examples:

```
<!--#include virtual="header.html"-->
<!--#exec cgi="counter.cgi"-->
<!--#echo var="DATE_LOCAL" -->
```

If the server is configured to parse and execute these directives and the user is able to insert his own data into the HTML code (via message board, guest book or some other method), the attacker can use this injection point to insert and execute his own directives. Attackers use SSI injection to access configuration files or execute code on the server side.

```
<!-- #include virtual="/web.config" -->
<!-- #exec cmd="/bin/ls /" -->
```

This vulnerability arises if there isn't implemented filtering of SSI commands before adding user's data to the page.

### 2.2.7. MX Injection

MX Injection [17.] is a vulnerability found in Web applications that communicate with mail servers. It is possible that such applications are vulnerable to injection of commands from mail protocols (IMAP and SMTP). If the attacker can alter IMAP and SMTP commands being transmitted to the mail server, the MX injection can occurs. This will depend on the validation filters that were implemented.

Attackers can use this vulnerability to perform actions on the mail server not normally permitted through the web interface. This can lead to information leakage, evasion of CAPTCHA or restrictions, sending SPAM, etc.

## 2.3. Malicious File Execution

Malicious File Execution vulnerability is another type of vulnerability that occurs due to not properly handling input data. When input data references some file and it is used with file or stream functions it can lead to remote code execution and system compromise.

Although, the problem is most common in PHP, other environments such as Java and .Net are also vulnerable to this attack.

The most widespread security issue in PHP is Remote code execution [22.]. The most dangerous functions prone to malicious file execution are include() and require(). The functions are used to load some code into the file before executing it. The vulnerable code in PHP can look like this:

```
$report = $_POST['report_name'];
include $report;
```

If parameter `report_name` is not validated, malicious code can be loaded and executed. Some other functions that can be used to exploit this vulnerability are:

```
fopen(), fsockopen(),
popen(), system(), ` (backtick operator),
eval(),
include_once(), require_once(),...
```

If the allow_url_fopen flag is set in php.ini, it is even possible to load files from different server.

If there's no validation for directory traversal when referencing file from user's input it is possible to access files outside the web application's root directory and execute them.

```
http://www.someapplication.com/execute.php?file=..\..\..\script.php
```

Attacker can use this vulnerability to load malicious code and execute it on the server. There are many reports of this vulnerability and most often they are found within some PHP applications: forums, blogs, CMS applications, etc.

Also, exploiting this vulnerability can lead to complete system compromise. The damage of the executed code depends on the privileges that Web application has on the remote system.

## 2.4. Insecure Direct Object Reference

When internal objects are referenced through URL, form parameters or cookies and no access control checking is implemented it is possible to access other objects that user shouldn't have permission to access. Internal objects can be files, directories or database records.

For example, let us assume that user's information is displayed through page:

```
http://www.someapplication.com/viewuser.php?id=xxx
```

Let us also assume that there's no access control implemented. Now, the malicious user can change an id parameter to some other value and access other user's information. Proper access control should allow user to access only hers/his information.

Another common problem is ability to reference local system files.

For example, if some parameter gets name of the file to be displayed and input is not checked against path traversal, it is possible to access system files such as /etc/passwd on Linux or boot.ini on Windows operating system.

```
http://www.someapplication.com/viewfile.php?name=../../../../../../../etc/passwd
```

Attacker can tamper with parameters and try to access other internal objects. It is possible to bypass authorization policies and access private and other user's information. When references point to keys in databases it is common that they are sequential, so an attacker can simply enumerate objects.

#### *Real-life example*

There was an attack in year 2000 to the Australian Taxation Office's GST Start Up Assistance. The legitimate user found out that he can access other company's details by tampering with the parameters in the URL. He managed to gather information on 17.000 companies and e-mailed them to report the vulnerability in the Web application [26.].

## 2.5. Cross Site Request Forgery (CSRF)

CSRF is the most widespread vulnerability today. Every Web application that is vulnerable to XSS is also vulnerable to CSRF, but absence of XSS vulnerability doesn't mean that application is still not vulnerable to CSRF. The vulnerability is also known by the names Session Riding, Hostile Linking and by the acronym XSRF.

CSRF is method of attack where victim without knowing sends requests to some web site that she/he is logged on and performs some unwanted operations. Victim does that because malicious site that she/he visited has malicious content which instructs client's browser to perform malicious requests. The simplest example is viewing page that has is source:

```
<img src="http://www.someforum.com/logout.php" />
```

The browser automatically requests URL in src attribute of img tag and if user is logged in to forum, she/he will be logged out. This happens because credentials that are used with vulnerable site are automatically submitted by the browser through for example session cookie. From the Web application's point of view, the request is completely valid and comes from a valid user. The difference is that it was performed without user's knowledge.

The main problem behind CSRF is that Web applications rely on credentials that are automatically submitted by browsers.

Logging out of some forum or web mail is the most harmless action that can be performed. CSRF can be used to perform more malicious operations, such as fund transfers, stealing sensitive data, etc. The only condition to perform some operations on vulnerable site is that the user is logged in on that site.

### *Real-life example*

There was a CSRF vulnerability found in Google that allowed stealing all Gmail contacts if user was logged in to Gmail [27.]. Google Docs has a script that runs a callback function, passing it user's contact list as an object. The script checks for authentication cookie, but it doesn't check if the request was made by Google Docs or some other application. The attacker was able to simply insert the script to his malicious site and with callback function parse read contact details from the object it received. This worked because browser sends cookies with authentication credentials automatically if the user is logged in. The proof-of-concept code of the callback function and the script URL are shown here:

```
<script type="text/javascript">

function google(a){
    var emails;
    emails = "<ol>"
    for(i=0;i<a.Body.Contacts.length;i++){
        mails += "<li>"+a.Body.Contacts[i].Email+"</li>";
     }
 emails += "</ol>"
 document.write(emails);
}
</script>

<script src = "http://docs.google.com/data/contacts?
out=js&show=ALL&psort=Affinity&callback=google&max=99999"> </script>
```

## 2.6. Information Leakage and Improper Error Handling

Information leakage and improper error handling are not the types of vulnerabilities that can be exploited, but they help and speed up the exploiting process. The leaked information can reveal information about underlying system, software distribution, version numbers and patch levels. Also, the information can contain names and locations of backup or temporary files, server names and IP addresses, usernames and test passwords. That kind of information can be found in forgotten developers comments or commented code or error messages. Automatic directory indexing/listing is a web server's function that list all the files and directories inside the requested directory if the default file is missing and it is also considered as information leakage.

Attackers can try to crash the application, so it generates an error message. If error message displays too much information to the user, it can be used to better understand how the application works. It can also reveal some internal object names, paths or other information.

One common example is error message displayed due to improper SQL, XPath or LDAP query. It can reveal information what database is in use, where in the query an error has occurred and even display to the user some part of the query. This information can be used to gain knowledge about internal queries and perform better injection attacks.

## 2.7. Broken Authentication and Session Management

Broken authentication deals with improper implementation of authentication mechanism and broken session management deals with belonging functions such as logout, timeout, secret question, password reset etc.

If the authentication mechanism is not well implemented it is possible to use that weakness to exploit the application. Often, faulty authentication mechanisms return different messages if the user entered invalid username, password or both. Example:

```
Invalid username.

Invalid password for the given username.

Invalid username and password.
```

This kind of messages reveal if the attacker has guessed the username so the attacker can first concentrate on finding the valid username and then, finding the valid password for the given username. This is much faster then guessing the username/password combination.

Also, application can have some hidden areas that doesn't use authentication due to belief that they are safe because they are not linked anywhere on the page. This kind of insufficient authentication can permit the user that guessed the location of hidden functions to use them without authenticating.

Weak password recovery can be implemented in such faulty way that allows the attacker to easily obtains, changes or recovers another user's password. Some password recovery system are guided by answering the user's secret question and there's no lockout policy defined so the attacker can brute force the answer and obtain the user's password. The hints that are sometimes displayed to the user can also sometimes reveal too much information and help in guessing or brute forcing the password.

Attacking session management vulnerabilities deals with the authorization of the user. The users are most often authorized using the session ids that are most often stored in session cookies. So the attacks most often targets these session ids.

If an algorithm for creating the session-id is weak, an attacker can guess or calculate other valid session-ids and access the site with compromised user’s privileges. The attack is known as credential/session prediction.

One of the attacks against session management is called session fixation attack. In this attack an attacker obtains sessionid from the target application and sends is to victim. Victim uses this sessionid to access the targeted application and logs on using his valid credentials. After that, the attacker can use this same sessionid to access the application under victim’s privileges.

Insufficient session expiration is a vulnerability that allows the user to reuse the old session id for authorization. This can improve the success probability of certain attacks when session ids have to be used.

As with insufficient authentication, the problem can also be in insufficient authorization when some hidden pages or functions are not properly protected and doesn't authorize user.

## 2.8. Insecure Cryptographic Storage

Web applications almost always have to use and store some sensitive data. If there's need for authentication and authorization process, there is need to store user's credentials and they should always be stored encrypted. Storing it in plain text is considered a security flaw. Even when encryption is implemented, various flaws can occur: wrong algorithm can be used: either weak or home made and even strong algorithms can be used insecurely.

If an attacker gets access to the database and the sensitive data is not encrypted, she/he can access it in its raw format. For example, if passwords are stored in clear text, the attacker can then immediately use these passwords to log on as different users. If the passwords were stored encrypted, that shouldn't be possible.

Storing passwords as hashes doesn't necessarily means that the passwords are secure. The attacker can use Rainbow tables to look up the original value for the given hash. Rainbow tables are pre-calculated tables that store string-hash combinations for every possible string using the characters from the given group and length. Calculating the table for all the possible combinations would need significant amount of time and space, but calculating the hashes for some common password strings or dictionaries can be done. Once the table is calculated, it can easily be searched to look up for the password. But if the password was hashed with some salt (that is, with some random value added to the password), the attacker would need to recalculate the entire Rainbow table so it prevents him to retrieve the password so easily.

## 2.9. Insecure Communications

Problems occur with failure to encrypt data or weak encryption during the transmitting sensitive information across the network.

Whenever transmitting sensitive data across the network, the data should be encrypted. The failure to do so is vulnerability and it makes possible the sensitive information leakage. Sniffing wireless traffic is much easier, as anybody can sniff it, but cabled networks must also used encryption because an attacker can install sniffer on compromised computer on the network.

Using weak encryption algorithms is also considered a problem.

An attacker can sniff network traffic and record all the conversation. If there's no encryption used wile transmitting sensitive data, the attacker would be able to understand it. It can be used to steal credentials during logon process, session cookies or some other information.

## 2.10. Failure to Restrict URL Access

Problem arises when application fails to restrict access to some URLs. Just not having the link anywhere in application to the protected source, don't necessary mean that the resource won't be found. An access control check must me implemented before requesting some restricted page or function.

An Attacker can guess paths to protected resources using the so called forced browsing.

# 3. Automated detection of vulnerabilities

This section will explain how the tools for automated scanning search for vulnerabilities in Web applications. In order to find details how they do that, only open source tools were examined. All the tools use black-box testing technique that scan the Web application from the outside and doesn't look into application's source code or has any information regarding the application's logic. Once again, the list of vulnerabilities is taken from the OWASP Top 10 2007 document which covers the most important ones:

A1. Cross Site Scripting (XSS)

A2. Injection Flaws

A3. Malicious File Execution

A4. Insecure Direct Object Reference

A5. Cross Site Request Forgery (CSRF)

A6.Information Leakage and Improper Error Handing

A7. Broken Authentication and Session Management

A8. Insecure Cryptographic Storage

A9. Insecure Communications

A10. Failure to Restrict URL Access

If some vulnerability can't be found using automated scanners, it will be explained why.

## 3.1. Cross Site Scripting (XSS)

Scanner needs to request some page and provide testing payload, search for that payload in response and conclude if the payload will execute. The easiest type of XSS to scan automatically is Reflected XSS because the payload is echoed back immediately. The problem with Stored XSS is that payload can occur later on some other page, so the scanner needs to keep track what payload was injected through which parameter and connect those two. In order to scan for DOM based XSS, the scanner has to have ability to parse Javascript. There are many tools that scan for XSS vulnerabilities but most of them search for only Reflected XSS, and only few search for DOM based XSS. In this section, the following tools will be examined: Gamja, Wapiti, w3af and Javascript XSS scanner.

### *Gamja – Reflected XSS*

Gamja is a command-line tool written in Perl that scan for XSS vulnerabilities and SQL injection flaws [30.]. It is still in early beta phase.

Gamja search only for Reflected XSS. The XSS vector that Gamja provides through request is simple:

```
">XSS_Check
```

If that vector occurs in response, it concludes there is a XSS problem. The problem with this technique is that this payload doesn't cover all cases when injection can occur and execute. It is more a sign that the payload was echoed back and that XSS might occur.

Gamja doesn't search for Persistent or DOM-based XSS.

### *Wapiti – Reflected and Stored XSS*

Wapiti is a Web application vulnerability scanner / security auditor written in Python. It performs black-box scans and acts like a fuzzer, injecting payloads to see if application is vulnerable [31.]. It searches for file handling errors, XSS injections and various injection flaws (SQL, LDAP, CRLF,...)

Wapiti search for both Reflected and Stored XSS. It constructs XSS vector in more complicated way that can later be used to find out from what page and parameter the vector originated.

```
<script>var wapiti_[0-9a-h]+_[0-9a-h]+=new Boolean\(\);</script>
```

The first group of characters behind '`wapiti_`' is hex encoded string of page that XSS vector originated from, and second group is hex encoded name of parameter that XSS vector originated. The tool claims there is a XSS vulnerability if the XSS vector is found in response.

### *w3af – Reflected, Stored and DOM based XSS*

W3af stands for Web Application Attack and Audit Framework. It is the most detailed tool so far and it search for many vulnerabilities [32.]. It also search for all three types of XSS.

In creating XSS vectors, it uses multiple variants of injections with single quotes, double quotes or no quotes at all:

```
<SCRIPT>alert2('RANDOMIZE')</SCRIPT>
javascript:alert('RANDOMIZE');
JaVaScRiPt:alert('RANDOMIZE');
javas\tcript:alert('RANDOMIZE');
<SCRIPT>a=/XSS/\nalert(a.source)</SCRIPT>RANDOMIZE
javascript:alert("RANDOMIZE");
JaVaScRiPt:alert("RANDOMIZE");
<SCRIPT>alert("RANDOMIZE")</SCRIPT>
javas\tcript:alert("RANDOMIZE");
```

The term `RANDOMIZE` is then transferred into some random number value.

The scanner doesn't only search for XSS vector in response, but it also analyses the response and tries to remove false positives. If XSS is not found for some parameter, the scanner even reports which filters were used to prevent it.

W3af also has a plugin that search for DOM-based XSS. It searches through HTML code for script tags that use one of predefined suspicious Javascript functions with DOM variables that can be controlled by users. The list of functions is:

```
document.write
document.writeln
document.execCommand
document.open
window.open
eval
window.execScript
```

The list of variables that might be controlled by user is:

```
document.URL
document.URLUnencoded
document.location
document.referrer
window.location
```

If theres a connection found between those two, w3af reports a DOM based XSS bug.

### *Javascript XSS Scanner – Reflected XSS*

Javascript XSS Scanner is GNUCITIZEN's project that demonstrates it is possible to write XSS scanner in Javascript and execute it in every browser [33.]. It also uses few simple XSS vectors:

```
/XSS_SCAN"><script>
/XSS_SCAN'><script>
```

and search for them in the response.

## 3.2. Injection Flaws

Scanning for injection flaws is in most cases the same. The scanner provides some string that is supposed to raise an error in the interpreter, and then it searches for some common error message in the response page. However, for the completeness of this document, each injection flaw described in section 2. will be examined separately.

Most automated scanners search only for SQL injection bugs, as they are most often in Web applications, then some search for LDAP and XPath and only few of them search for other injection types.

### *SQL Injection*

Almost every tool that search for vulnerabilities in Web applications search for SQL injection. The approach in all tools is the same, it only differs in payloads used in query and in error messages that the scanner looks for.

**Gamja** tests Web applications for all injection flaws in same way. It appends a single quote and some text to the parameter and then search for common error expressions in response:

```
Warning:
JDBC
SQL syntax
ODBC
'SQL
error
```

This error expressions are too short and can come up with many false positives.

**Wapiti** also have only one payload for all the injection tests:

```
\xbf'"(
```

In parsing the response, it searches for common error messages and decides what type of injection is it (SQL, XPath, LDAP) and if it's SQL injection, it decides what underlying database is used (MySQL or MSSQL). First two errors it connects with MySQL, the third with MSSQL and the forth is a general one.

```
You have an error in your SQL syntax
supplied argument is not a valid MySQL
[Microsoft][ODBC Microsoft Access Driver]
java.sql.SQLException: Syntax error or access violation
```

**W3af** uses the same approach but it has larger database of common errors and it covers more databases. It uses the following payload: `d'z'0`. Given error messages that w3af search for are in a form `error_ message : underlying_database`:

```
[IBM][CLI Driver][DB2                                          : IBM db2 database
[SQL Server]                                                   : Microsoft SQL database
[Microsoft][ODBC SQL Server Driver]                            : Microsoft SQL database
[SQLServer JDBC Driver]                                        : Microsoft SQL database
[SqlException                                                  : Microsoft SQL database
'80040e14'                                                     : Microsoft SQL database
mssql_query()                                                  : Microsoft SQL database
odbc_exec()                                                    : Microsoft SQL database
ORA-0                                                          : Oracle database
ORA-1                                                          : Oracle database
PostgreSQL query failed                                        : PostgreSQL database
supplied argument is not a valid PostgreSQL result             : PostgreSQL database
supplied argument is not a valid MySQL                         : MySQL database
mysql_fetch_array()                                            : MySQL database
mysql_                                                         : MySQL database
on MySQL result index                                          : MySQL database
You have an error in your SQL syntax;                          : MySQL database
MySQL server version for the right syntax to use               : MySQL database
Incorrect syntax near                                          : MySQL database
com.informix.jdbc                                              : Informix database
Dynamic Page Generation Error:                                 : Informix database
Microsoft JET Database Engine error                            : Microsoft Access database
Microsoft OLE DB Provider for ODBC Drivers error               : Microsoft SQL database
[MySQL][ODBC                                                   : MySQL database
Column count doesn't match                                     : MySQL database
java.sql.SQLException                                          : Java connector
<b>Warning</b>:  ibase_                                        : Interbase database
```

However, this approach of detecting common error messages in response fails if the SQL errors aren't echoed back to the user. Another approach can be used and it is called Blind SQL injection.

W3af constructs two sets of payloads, one where true condition is added to the SQL statement, and one where false condition is added. Then it compares responses and tries to see if added conditions made any difference to he page. Both sets contains statement with single quotes, double quotes and no quotes, in order to successfully inject SQL code in different databases and SQL statements.

```
%i OR %i=%i
%i' OR '%i'='%i
%i" OR "%i"="%i
```

For comparing responses, there is also more then one method.

### *XPath Injection*

Scanning for XPath injection is done the same way as scanning for SQL or LDAP injections. The payload is passed to the parameters that should raise an error and then the response HTML is parsed and searched for some common expression that is sign of XPath error. Wapiti uses the same payload that is used for other injection types:

```
\xbf'"(
```

and it search for the occurrence of the string '`XPathException`'.

W3af also uses only one payload:

```
d'z'0
```

but it search for several error messages that may indicate there is XPath injection vulnerability:

```
Unknown error in XPath
org.apache.xpath.XPath
A closing bracket expected in
An operand in Union Expression does not produce a node-set
Cannot convert expression to a number
Document Axis does not allow any context Location Steps
Empty Path Expression
Empty Relative Location Path
Empty Union Expression
Expected ')' in
Expected node test or name specification after axis operator
Incompatible XPath key
Incorrect Variable Binding
libxml2 library function failed
libxml2
xmlsec library function
xmlsec
```

Default error messages are sometimes not displayed to the user and they are substituted by custom non descriptive error messages or even blank pages. Then, the same principle as in detecting blind SQL injection could be used.

### *LDAP Injection*

Scanning for LDAP injection is done passing payload to the parameters that would raise an error if transferred to LDAP query. The error is raised because the payload constructs invalid LDAP request. Scanners use mixture of odd characters that makes the query invalid. Wapiti uses the same payload that is used for other injection types:

```
\xbf'"(
```

and w3af uses two different payloads, but the method is the same:

```
^(#$!@#$
)(()))
```

Both tools then parse the response page and look for one of the following strings:

```
supplied argument is not a valid ldap
javax.naming.NameNotFoundException
```

If the string is found, it is a sign that LDAP error has occurred and that the tested parameter is vulnerable to LDAP injection.

### *OS Commanding*

Scanning for OS Commanding doesn't distinguish much from other injection flaws scanning. W3af defines two commands that are combined with several piping characters and passed as parameter values to the application. The whole list of combinations is:

```
/bin/cat /etc/passwd
&/bin/cat /etc/passwd
|/bin/cat /etc/passwd
;/bin/cat /etc/passwd
`/bin/cat /etc/passwd`
type C:\\boot.ini
&type C:\\boot.ini
|type C:\\boot.ini
;type C:\\boot.ini
```

Then, in response HTML, it is searched for one of the following strings:

```
root:x:0:0:
daemon:x:1:1:
:/bin/bash
:/bin/sh
[boot loader]
default=multi(
[operating systems]
eval()'d code</b> on line <b>
Cannot execute a blank command in
Fatal error</b>:  preg_replace
```

The occurrence of one of these strings suggests that the commands were executed and that the output was echoed back t the user.

### *SSI Injection*

Scanning for SSI injection includes providing a few SSI directives to the Web application and detection if those directives were executed and if content was included to the page. W3af uses following two directives:

```
<!--#include file=\"/etc/passwd\"-->
<!--#include file=\"C:\\boot.ini\"-->
```

Parsing the response search for some common strings that would appear in this configuration files:

```
root:x:0:0:
daemon:x:1:1:
:/bin/bash
:/bin/sh
[boot loader]
```

```
default=multi(
[operating systems]
```

Finding one of these strings suggests that the directives were actually executed and the content was included into the page.

### *CRLF Injection*

Detecting CRLF Injection is a straight-forward process. To every parameter that is tested, two special characters are appended : carriage-return (ASCII code %0d - \r) and line feed (ASCII code %0a - \n) and then, new header with parameter is added. In the response, it is checked if there is new header included in the response. Wapiti uses the following payload:

http://www.google.fr\r\nWapiti: version 1.1.6

and w3af uses:

w3af\r\nVulnerable: Yes

w3af goes one step further and checks not only if the new header is present but also if the header value matches inserted one.

### *MX Injection*

Searching for MX injection is similar to searching for other injection vulnerabilities. Few strings are provided that are intended to raise errors in the backend and then parsers try to find evidences of these errors in HTML. W3af searches for the following strings:

```
Unexpected extra arguments to Select
Bad or malformed request
Could not access the following folders
To check for outside changes to the folder list go to the folders page
A000
A001
Invalid mailbox name
```

## 3.3. Malicious File Execution

The automated approach doesn't work well with malicious file execution because the tools have problems with identifying parameters used for attack.

The scanners cannot know if some parameter references some file on the file system so they have to suppose that every parameter is vulnerable to this attack. Database with paths to most common files on both Windows and Linux systems is used to try to access those files.

W3af search for remote file inclusion flaw. First. it creates random php script and opens the web server with that script hosted. Then, it tries to include this script via vulnerable parameter and checks if the script has executed.

## 3.4. Insecure Direct Object Reference

Scanners are good only at detecting path traversal bugs, so in that case, they can detect insecure reference to local system files. If there's filtering and access control implemented, there is still possibility that malicious user can access some files in current directory that he/she shouldn't have be able to, but the scanner cannot know what are the access rules.

When detecting path traversal, scanners use predefined paths to default system files and then parse the response and search for some strings that occur in those files. W3af uses following list when attempting to access system files:

```
../../../../../../../../etc/passwd
/etc/passwd
/etc/passwd\0
../../../../../../../../../boot.ini\0
C:\\boot.ini
C:\\boot.ini\0
```

Strings that are searched for in response are :

```
root:x:0:0:
daemon:x:1:1:
:/bin/bash
:/bin/sh
[boot loader]
default=multi(
[operating systems]
java.io.FileNotFoundException:
fread():
for inclusion (include_path=
Failed opening required
```

Wapiti uses similar lists to detect insecure local file references. Finding some of the above strings is the sign that the system file was accessed and displayed to the user.

When trying to detect insecure references to other objects such as database keys, the scanners can use parameter tampering but they cannot know if the accessed object should be protected from the user or not. For example, if the user has right to access object `xxx` from the database, the scanner cannot know if objects `yyy` and `zzz` should be protected or not.

```
http://www.someapplication.com/viewItem.php?id=xxx
http://www.someapplication.com/viewItem.php?id=yyy
http://www.someapplication.com/viewItem.php?id=zzz
```

The scanner can use some heuristics to try to guess that, but it can result in many false positives.

## 3.5. Cross Site Request Forgery (CSRF)

It is very hard for automated tools to detect CSRF. That is because the nature of the attack is to force the user to send the legitimate request to some other resource. The credentials to access the resource are sent automatically by the user's browser. Scanners can find occurrences to some extern resources but cannot know if they are permitted or not.

## 3.6. Information Leakage and Improper Error Handling

The tools have trouble with understanding the meaning of the messages, but can be configured to search for some common words or phrases that indicate information leakage.

Scanners do well in finding these types of vulnerabilities.

Scanners help in automating the source review of every page in order to find comments that leak to much information. W3af search for occurrence of following words/strings:

```
'user', 'pass', 'xxx', 'fix', 'bug', 'broken', 'oops', 'hack', caution', 'todo',
'note', 'warning', '!!!', '???', 'shit'
```

When trying to find too descriptive error messages it also uses the database with common error messages or fragments of HTML code. It can also use these messages to find out on what system is the application running and what database does it use.

Some scanners also search for leakage of private IP addresses and SVN signatures.

## 3.7. Broken Authentication and Session Management

Flaws in authentication and session management are very difficult to find for automated scanning tools. The scanners can not really know if the accessed resources should be protected by authentication and authorization mechanisms or not. If there is implemented an authentication, the scanners have trouble authenticating and often need human intervention (providing username ans password).

When the Web application returns session ID to the user, the scanner must locate it within all the parameters returned to the user. This can be done using some heuristics but it also isn't simple. Once the scanner know what is session ID, it can try to change it and hijack some other's session. This is again difficult, because the scanner can not know if the attack is successful. Changing the session ID to match some other user's session can result in very similar page, very different or somewhere in between.

The only good thing that scanners can do here is analyzing of the received session IDs for relative predictability. However, if they succeed to predict the next session ID, it leaves it in difficulty to tell if the attack was successful.

## 3.8. Insecure Cryptographic Storage

Vulnerability scanning tools cannot verify if encryption is used when storing sensitive data. From the outside view, there is no possibility to know how the data is stored, if there are weaknesses in encryption and if there is encryption used at all. Even the human intervention can not gain any more information here, and the only way to know this is to look to source code and database on the server.

## 3.9. Insecure Communications

The scanning tools can detect if SSL is used between client and server but cannot verify if Web application access some other server in the backend and if that communication is secure.

The easiest approach to test whether secure communication is used is to check what protocol was used (HTTP or HTTPS) and what port was used (80, 443 or some other).

Detecting communication between backend servers and tested application is not possible and should be manually tested.

W3af checks if the protected resources can also be accessed without using the encryption. For every page that is accessed via secure protocol (that is HTTPS), it tries to access the same page using the non encrypted protocol (HTTP).

## 3.10. Failure to Restrict URL Access

Tools cannot know which page should be accessible by which user and therefore cannot verify if there exist a failure to protect a page or function.

Scanners do try to find hidden URLs and functions but again, they cannot know if these resources really need to be protected by access control or not. In order to find hidden resources, they use various techniques:fuzzing directory and file names, using dictionary lists, trying to find back-up files, folders, etc.

For example, w3af takes each page's name and appends different file extensions in order to find forgotten compressed or backup files.

When trying to guess parameter values, it takes few known values for parameters and queries Google Sets [34.] to retrieve other predicted items for the given set. For example, if there are URLs found in application:

```
http://www.someapplication.com/index.asp?color=blue
http://www.someapplication.com/index.asp?color=red
```

w3af takes values from color parameter and queries Google Sets which will return other similar items: `black, white, green`, etc.

Another algorithm to force browsing is changing the numbers found in parameter names or values. Incrementing or decrementing them can result in finding hidden pages that weren't linked anywhere on the web site.

# 4. Conclusion

In this paper, there are described most common vulnerabilities found in Web applications. Some of them are general and can be found in most of Web applications and some are very specific. All vulnerabilities are considered risk to the company who owns the Web application and exploiting them can lead to harm and financial loss.

In order to prevent such attacks companies perform security audits, where they can use source code audit, or black-box scan or combination of those two. In this paper, it is given description how the black-box scanners perform their scan and what are they good for.

It was found out that, indeed, tools for automated scanning can help and ease the process of security audit, but there are areas where they have difficulty and where other approaches should be used. Results from such tools can be used as an indication where should the penetration tester take a deeper look.

Also, it was found out that designing and building a modular scanner is a complicated process and that requires much consideration around side functionalities and does not solely rely on a process of vulnerability identification.

# 5. References


1. The Open Web Application Security Project (OWASP), http://www.owasp.org/
2. OWASP Top 10 2007: The ten most critical Web application security vulnerabilities, OWASP Foundation
3. S. Christey, R. A. Marti, Vulnerability Type Distributions in CVE, http://cwe.mitre.org/documents/vuln-trends/index.html, (15/07/07)
4. Web Application Security Consortium – Security Statistics, http://www.webappsec.org/projects/statistics/, (15/07/07)
5. A. Klein, DOM Based Cross Site Scripting or XSS of the Third Kind, http://www.webappsec.org/projects/articles/071105.shtml
6. J. Grossman, Cross-Site scripting worms and viruses, WhiteHat Security, 2006
7. Samy, MySpace Worm Explanation, http://namb.la/popular/tech.html
8. R. Valotta, Nduja Connection Worm, http://rosario.valotta.googlepages.com/
9. Benjamin Flesch, This is the first Weblog XSS Worm, http://mybeni.rootzilla.de/mybeNi/2007/this_is_the_first_weblog_xss_worm/
10. B. Rios, N. McFeters, Remote Command Exec (FireFox 2.0.0.5), http://xs-sniper.com/blog/remote-command-exec-firefox-2005/
11. K. Spett, SQL Injection, SPI Dynamic, http://www.spidynamics.com
12. SQL Injection, OWASP, http://www.owasp.org/index.php/SQL_injection
13. SPI Labs, Blind SQL Injection, SPI Dynamics
14. SPI Labs, XPath Injection, SPI Dynamics, http://www.webappsec.org/projects/threat/classes/xpath_injection.shtml
15. S. Faust, LDAP Injection, SPI Dynamics, http://www.webappsec.org/projects/threat/classes/ldap_injection.shtml
16. A. Klein, HTTP Response Splitting Whitepaper, Sanctum Inc., 2004
17. V. A. Diaz, MX Injection - Capturing and Exploiting Hidden Mail Servers, Internet Security Auditors, 2006.
18. OS Commanding, Web Application Security Consortium, http://www.webappsec.org/projects/threat/classes/os_commanding.shtml
19. Server Side Includes, Wikipedia, http://en.wikipedia.org/wiki/Server_Side_Includes
20. SSI Injection, Web Application Security Consortium, http://www.webappsec.org/projects/threat/classes/ssi_injection.shtml
21. Using xp_cmdshell, Larsen G.A., http://www.databasejournal.com/features/mssql/article.php/3372131
22. W3Schools, http://www.w3schools.com/php/php_includes.asp
23. OWASP File System, http://www.owasp.org/index.php/File_System#Includes_and_Remote_files

24. OWASP PHP Top 5, http://www.owasp.org/index.php/PHP_Top_5#P1:_Remote_Code_Execution
25. OWASP Testing for DT, http://www.owasp.org/index.php/Testing_for_Directory_Traversal
26. GST Assist attack details, ABC, http://www.abc.net.au/7.30/stories/s146760.htm
27. Gmail contact flaw, http://betterexplained.com/articles/gmail-contacts-flaw-overview-and-suggestions/
28. Threat Classification, Web Application Security Consortium, http://www.webappsec.org/projects/threat/classes_of_attack.shtml
29. J. Grossman, Automated scanning vs. the OWASP Top 10, WhiteHat Security, 01/2007
30. Gamja: Web vulnerability scanner,http://sourceforge.net/projects/gamja
31. Wapiti, Web application vulnerability scanner, http://wapiti.sourceforge.net/
32. w3af - Web Application Attack and Audit Framework, http://w3af.sourceforge.net/
33. Javascript XSS Scanner, GNUCITIZEN, http://www.gnucitizen.org/projects/javascript-xss-scanner/
34. Google Sets, http://labs.google.com/sets