

## ABSTRACT

This white paper stresses the need for effective security testing and highlights the essence of path traversal, SQL injection and cross site scripting. There are several challenges faced while assuring the security of an application.

This paper aims to identify the prevalence and probability of different vulnerability classes and compare testing methodologies against the types of vulnerabilities they are likely to identify.

## INTRODUCTION

The Internet has brought about many changes in the way organizations and individuals conduct business, and it would be difficult to operate effectively without the added efficiency and communications brought about by the Internet.

The motivation for targeting web servers includes:

1. Web servers are always online where as home computer systems are often shutdown when not in use.
2. Web servers have more network bandwidth than home computer users. This essentially is a Quality of Service metric where commercial web servers are guaranteed specific amounts of network bandwidth usage whereas home computer users typically have much less bandwidth. Additionally, home user network traffic is oftentimes throttled which would make their DDoS attack traffic less.
3. Web servers have more horse power then home computers. The number of CPUs, RAM, etc. means that commercial servers can generate much more network DDoS traffic then home computer systems.
4. Web servers are less likely to be blacklisted by ISP vs. home computer systems. This means that web server zombies will be online, sending traffic much longer than home computers.

### What is Web Application Security Testing?

The black box testing method is a technique for hardening and penetration testing Web applications where the source code to the application is not available to the tester. It forces the tester to look at the Web application from a user's perspective (and therefore, an attacker's perspective). The tester, at first, tries to get a 'feel' for the application and learn its expected behavior. The term black box refers to this Input/Unknown Process/Output approach to testing.

Any strange behavior on the part of the application, in response to strange inputs, is certainly worth investigating as it may mean the developer has failed to validate inputs correctly!

Application vulnerabilities account for the vast majority of web server attacks. OS and web server vendors have worked hard to make their products less exploitable but, in general, application developers tend to be under trained in security. To be fair, most managers are more focused on delivering applications on time than with developing and testing the security defense of the application against some "theoretical" attack.

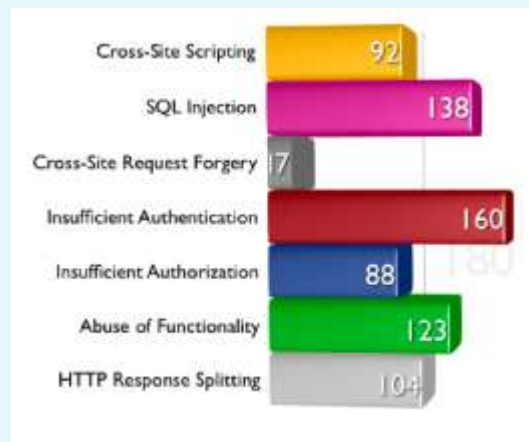


Figure 1

Application vulnerabilities are often caused by incorrect permission setting, scripting errors, design faults, inadequate input parameter checking, and by coding errors in general.

Back-end databases, such as MS-SQL, MySQL, PostGres, and Oracle, are frequently the targets of intruder attacks because they often directly interface with front-end web servers. Although MS-SQL hasn't had major system vulnerability in over three years, many other database suites, including Oracle and MySQL, have.

More often, application-level vulnerabilities (SQL injection attacks, scripting subsystem holes, and so on) allow the attacker to compromise the back-end database server and then use the data connection to compromise the web server.

Web servers use a number of network, transport, and session protocols. The two most popular protocols used are Hypertext Transfer Protocol (HTTP) and HTTP over SSL (HTTPS). However, it would not be an exaggeration to say that a web server can utilize dozens of other protocols (and file formats) to serve up a standard application. Each protocol and the subsystem that interacts with the client is a potential attack vector.

Even when the protocol doesn't have known public vulnerabilities, web developers can implement them insecurely. It is not uncommon for Web sites claiming to have HTTPS encryption and authentication to send all resulting data streams in clear-text and unauthenticated.

### SQL Injection

Many Web application developers (regardless of the environment) do not properly strip user input of potentially "nasty" characters before using that input directly in SQL queries. Depending on the back-end database in use, SQL injection vulnerabilities lead to varying levels of data/system access for the attacker. It may be possible to not only manipulate existing queries, but to UNION in arbitrary data, use sub selects, or append additional queries. In some cases, it may be possible to read in or write out to files, or to execute shell commands on the underlying operating system.

#### Locating SQL Injection Vulnerabilities

Often the most effective method of locating SQL injection vulnerabilities is by hand - studying application inputs and inserting special characters. With many of the popular backend, informative errors pages are displayed by default, which can often give clues to the SQL query in use: when attempting SQL injection attacks, you want to learn as much as possible about the syntax of database queries.

The tester attempts to elicit exception conditions and anomalous behavior from the Web application by manipulating the identified inputs - using special characters, white space, SQL keywords, oversized requests, and so forth. Any unexpected reaction from the Web application is noted and investigated. This may take the form of scripting error messages (possibly with snippets of code), server errors (HTTP 500), or half loaded pages.

#### Authentications bypass using SQL injection

This is one of the most commonly used examples of SQL injection vulnerability, as it is easy to understand for non-SQL-developers and highlights the extent and severity of these vulnerabilities. One of the simplest ways to validate a user on a Web site is by providing them with a form, which prompts for a username and password. When the form is submitted to the login script (eg. login.asp), the username and password fields are used as variables within an SQL query.

In this scenario, no sanity or validity checking is being performed on the user and pass variables from our form inputs. The developer may have client-side (e.g. JavaScript) checks on the inputs, but any attacker who understands HTML can bypass these restrictions.

#### PHP and MySQL Injection

A vulnerable PHP Web application with a MySQL backend, despite PHP escaping numerous 'special' characters (with Magic Quotes enabled), can be manipulated in a similar manner to the above ASP application. MySQL does not allow for direct shell execution; however in many cases it is still possible for the attacker to append arbitrary conditions to queries, or use UNIONS and sub selects to access or modify records in the database.

## Code and Content Injection

What is code injection? Code injection vulnerabilities occur where the output or content served from a Web application can be manipulated in such a way that it triggers server-side code execution. In some poorly written Web applications that allow users to modify server-side files (such as by posting to a message board or guestbook) it is sometimes possible to inject code in the scripting language of the application itself. This vulnerability hinges upon the manner in which the application loads and passes through the contents of these manipulated files - if this is done before the scripting language is parsed and executed, the user-modified content may also be subject to parsing and execution.

## Path Traversal and URIs

This category of attacks exploits various path vulnerabilities to access files or directories that are not intended to be accessed. This attack works on applications that take user input and use it in a "path" that is used to access a filesystem. If the attacker includes special characters that modify the meaning of the path, the application will misbehave and may allow the attacker to access unauthorized resources. This type of attack has been successful on web servers, application servers, and custom code.

A common use of Web applications is the act as a wrapper for files of Web content, opening them and returns in them wrapped in chunks of HTML. Once again, sanity checking is the key. If the variable being read in to specify the file to be wrapped is not checked, a relative path can be entered.

<http://www.example.com/index.php?file=../../etc/passwd>

This request would return the contents of /etc/passwd unless additional stripping of the path character (..) had been performed on the file variable.

This problem is compounded by the automatic handling of URIs by many modern Web scripting technologies, including PHP, Java and Microsoft's .NET. If this is supported on the target environment, vulnerable applications can be used as an open relay or proxy:

<http://www.example.com/index.php?file=http://www.google.com/>

This flaw is one of the easiest security issues to spot and rectify, although it remains common on smaller sites whose application code performs basic content wrapping. The problem can be mitigated in two ways. First, by implementing an internal numeric index to the documents or, by using files named in numeric sequence with a static prefix and suffix. Secondly, it can be done by stripping any path characters such as [^.] which attackers could use to access resources outside the applications directory tree.

Most web sites restrict user access to a specific portion of the file-system, typically called the "web document root" or "CGI root" directory. These directories contain the files intended for user access and the executable necessary to drive web application functionality. To access files or execute commands anywhere on the file-system, Path Traversal attacks will utilize the ability of special-characters sequences.

## Cross Site Scripting Attacks

Cross Site Scripting attack (a form of content-injection attack) differs from the many other attack methods in that it affects the client-side of the application (i.e. the user's browser). Cross Site Scripting (XSS) occurs wherever a developer incorrectly allows a user to manipulate HTML output from the application - this may be in the result of a search query, or any other output from the application where the user's input is displayed back to the user without any stripping of HTML content.

Basically, an attacker inputs a malicious script into a web site. This can be in a forum, comment section, or any other input area. When victims visit that web site, they only need to click on that script to start the exploit.

A few facts about cross-site scripting attacks that you should be aware of are:

- Every month roughly 10-25 XSS holes are found in commercial products and advisories are published explaining the threat.
- Websites that use SSL (https) are in no way more protected than websites that are not encrypted. The web applications work the same way as before, except the attack is taking place in an encrypted connection.

- XSS attacks are generally invisible to the victim.
- All Web servers, application servers, and Web application environments are susceptible to cross-site scripting.

A simple example of XSS can be seen in the following URL:

<http://server.example.com/browse.cfm?categoryID=1&name=Books>

In this example the content of the 'name' parameter is displayed on the returned page. A user could submit the following request:

<http://server.example.com/browse.cfm?categoryID=1&name=<h1>Books>

If the characters < > are not being correctly stripped or escaped by this application, the "<h1>" would be returned within the page and would be parsed by the browser as valid html. A better example would be as follows:

[http://server.example.com/browse.cfm?categoryID=1&name=<script>alert\(document.cookie\);</script>](http://server.example.com/browse.cfm?categoryID=1&name=<script>alert(document.cookie);</script>)

In this case, we have managed to inject JavaScript into the resulting page. The relevant cookie (if any) for this session would be displayed in a popup box upon submitting this request.

This can be abused in a number of ways, depending on the intentions of the attacker. A short piece of JavaScript to submit a user's cookie to an arbitrary site could be placed into this URL. The request could then be hex-encoded and sent to another user, in the hope that they open the URL. Upon clicking the trusted link, the user's cookie would be submitted to the external site. If the original site relies on cookies alone for authentication, the user's account would be compromised.

In most cases, XSS would only be attempted from a reputable or widely-used site, as a user is more likely to click on a long, encoded URL if the server domain name is trusted. This kind of attack does not allow for any access to the client beyond that of the affected domain (in the user's browser security settings).

### *Risks Associated with Cross-Site Scripting*

Attackers are lured to XSS exploits because how easy they are to perform, but they also know to follow the money. Attacking a web site through a cross-site scripting vulnerability can be quite profitable for the attacker who knows how to harness this type of exploit.

Without proactive Web application security in place to stop XSS attacks, you leave your site vulnerable to:

- User accounts being stolen through session hijacking (stealing cookies) or through the theft of username and password combinations
- The ability for attackers to track your visitors web browsing behavior infringing on their privacy
- Abuse of credentials and trust
- Keystroke logging of your site's visitors
- Misuse of server and bandwidth resources
- The ability for attackers to exploit your visitor's browser
- Data theft
- Web site defacement and vandalism
- Link injections
- Content theft Web sites that have been exploited using XSS attacks have also been used to:
- Probe the rest of the intranet for other vulnerabilities
- Launch Denial of Service attacks
- Launch Brute Force attacks

## ENDING NOTES

Organizations should protect web applications by deploying web application firewalls that inspect all traffic flowing to the web application for common web application attacks, including but not limited to Cross-Site Scripting, SQL injection, command injection, and directory traversal attacks. For applications that are not web based, deploy specific application firewalls if such tools are available for the given application type.

Again, web application firewalls can be used as a tactical remediation tool to help organizations reduce their time-to-fix metric of fixing identified vulnerabilities by acting as a virtual patch. The most widespread vulnerabilities are Cross-Site Scripting, Information Leakage, SQL Injection, Insufficient Transport Layer Protection, Fingerprinting, HTTP Response Splitting. As a rule, Cross-Site Scripting, SQL Injection and HTTP Response Splitting vulnerabilities are caused by design errors, while Information Leakage, Insufficient Transport Layer Protection and Fingerprinting are often caused by insufficient administration (e.g., access control).

## REFERENCES

1. <http://projects.webappsec.org/w/page/13246989/Web-Application-Security-Statistics>
2. <http://www.aldeid.com/index.php/WackoPicko/Directory-Traversal>
3. <http://programming4.us/security/614.aspx>