File Disclosure

Outline

- File Disclosure
  - Impact and Overview
  - Paths 101
  - Path traversal attacks
  - Fixes
- Server-Side Request Forgery

File Disclosure

- A file disclosure is the **impact of certain vulnerabilities**
- As the name suggests, it consists of the ability to **disclose/leak important files from a server**
- Because it is an impact, there are **multiple classes of vulnerabilities** that **lead to file disclosure**
    - For example, remote code execution is another type of vulnerability that could results in a file disclosure

File Disclosure

- Files inside a server are critical information:
  - In many applications, users-uploaded files are the sensitive information that the application is protecting
  - The disclosure of such files can be a violation of the site policy

File Disclosure

- It is also possible to **steal configuration files** from the webserver **which might contain critical information items**
  - *Database configuration files* often contain the credentials to access the database
  - Files like the *tomcat-users.xml* contain the credentials to access the tomcat manager
  - Files like *flask configuration* or *web.config* in a .net application contain the secret used to sign the session

File Disclosure

- Finally, it is possible to **steal the source code** of the web application
  - For some business, the source code of the web application is its **product/asset**
  - An attacker in possession the source code is more effective
    - It is easier for the attacker to find other vulnerabilities, especially if the application was developed according to a *security by obscurity* model

File Disclosure

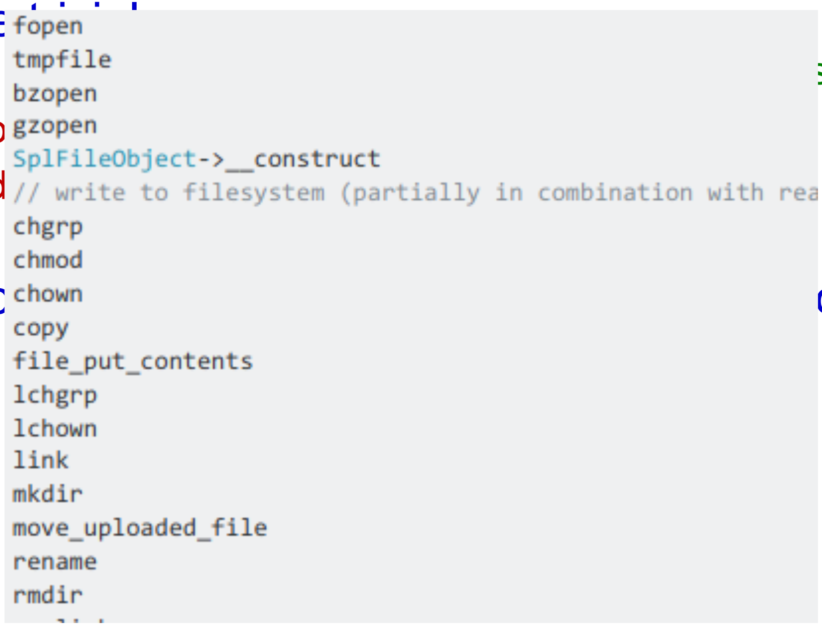
- How can a web app disclose internal files?
  - Basically, **everything that works with files can lead to a file disclosure vulnerability**
  - There are standard sinks, and some of them are a trivial
  - If a user-controlled input manages to go inside these sinks, the web app is at risk

File Disclosure

- Some sinks are trivial...
  - Every function in every programming language that manages files
    - Every flavor of **open/fopen** in every language
    - Flask **send_file**
    - …
- It is also possible to leak files if the web app suffers from **code execution**

File Disclosure

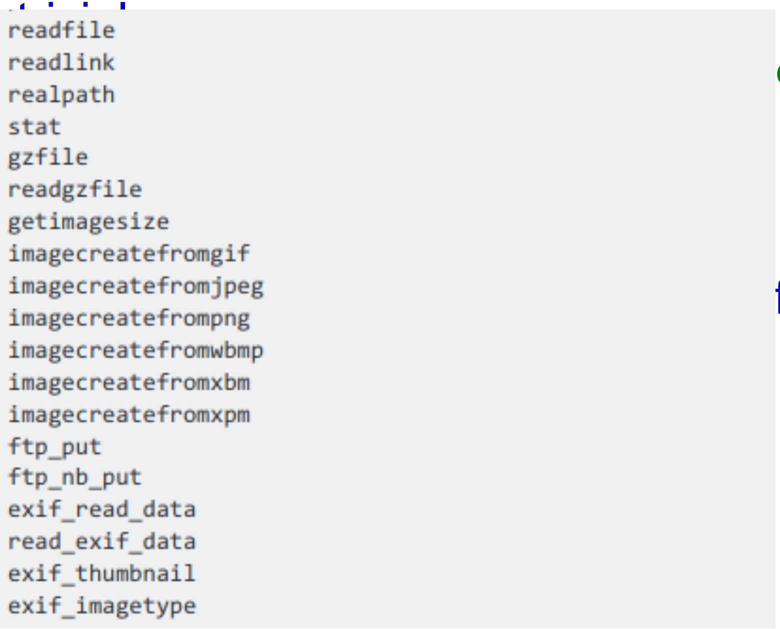
- Some sinks are
  - Every function … s files
    - Every flavo
    - Flask **send**
    - …
- It is also possib … om **code execution**



```
fopen
tmpfile
bzopen
gzopen
SplFileObject->__construct
// write to filesystem (partially in combination with rea
chgrp
chmod
chown
copy
file_put_contents
lchgrp
lchown
link
mkdir
move_uploaded_file
rename
rmdir
```

File Disclosure

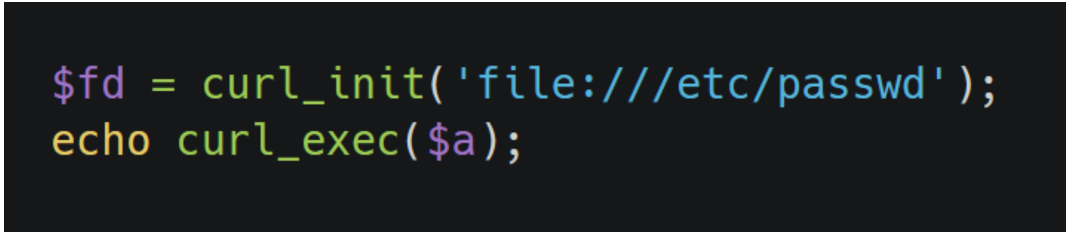
- Some sinks are
  - Every function es files
    - Every flavo
    - Flask **send_**
    - …
- It is also possib from **code execution**



```
readfile
readlink
realpath
stat
gzfile
readgzfile
getimagesize
imagecreatefromgif
imagecreatefromjpeg
imagecreatefrompng
imagecreatefromwbmp
imagecreatefromxbm
imagecreatefromxpm
ftp_put
ftp_nb_put
exif_read_data
read_exif_data
exif_thumbnail
exif_imagetype
```

File Disclosure

- Other sinks are less trivial
    - **cURL** is used as a http client. But it can also be used to open files



```
$fd = curl_init('file:///etc/passwd');
echo curl_exec($a);
```

File Disclosure

- It’s sometimes possible to leak important files just because they are publicly accessible
  - *.git* directory exposed
    - If you make your git directory open to the internet, everyone will be able to dump all files inside it
  - Web-server misrouting
    - It’s sometimes possible to trick a web server to return a .php file as an image...

Paths 101

- Let us focus on what happens if a user-controlled input finds a way to an open-like function
- We first need to understand few things about how paths work

Paths 101

- An **absolute path** is a path that describes the location of a file regardless of the working directory

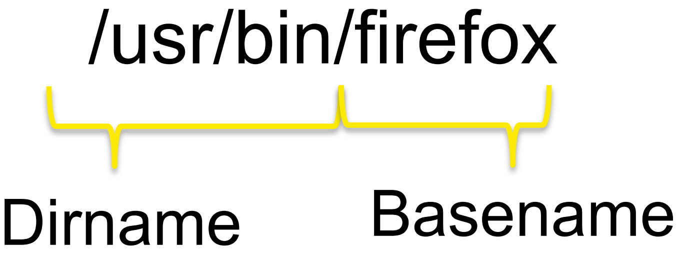
```
/etc/passwd
```

- A **relative path** is a path that describes the location of a file starting from the working directory

```
foo/bar
```

Paths 101

- Paths are composed by a **dirname** and a **basename**
  - The **dirname** is the portion of the path up to the last /
  - The **basename** is the portion of the path after the last /



/usr/bin/firefox

Dirname

Basename

Paths 101

- Every directory has two special subdirectories:
  - The **current directory**, whose name is .

/foobar/./ → /foobar/

  - And the **parent directory**, whose name is ..

/foobar/../baz → /baz

- The parent directory is useful for file disclosure because it permits to access deeper directories inside the file system

Paths 101

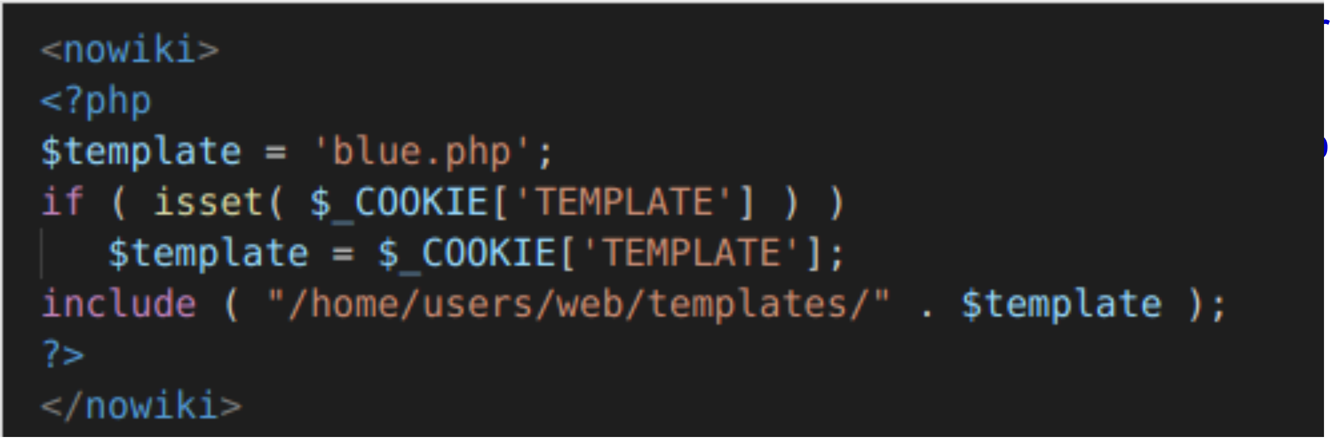
- A path in its **shortest form** is called **normalized**
- For example:
  - */foo/bar* is normalized, there is no way to make it shorter
  - *//foo/bar* is not normalized, /foo/bar is shorter
  - */foo/./bar* is not normalized, /foo/bar is shorter
- What about /foo/test/../bar?

Paths 101

- What about /foo/test/../bar?
- Its shortest form would be /foo/bar, but what happens if /foo/test/ does not exist?
  - If the path is normalized before opened, then everything is fine: we can access /foo/bar without any problem
  - If the path is not normalized, then the open would fail because /foo/test/ does not exist, and so ..

Path Traversal

- **Path traversal** is a vulnerability that leads to a file disclosure
- It happ an equivalen
- If there uld inject path



```
<nowiki>
<?php
$template = 'blue.php';
if ( isset( $_COOKIE['TEMPLATE'] ) )
   $template = $_COOKIE['TEMPLATE'];
include ( "/home/users/web/templates/" . $template );
?>
</nowiki>
```

Path Traversal

- Few cases might happen:

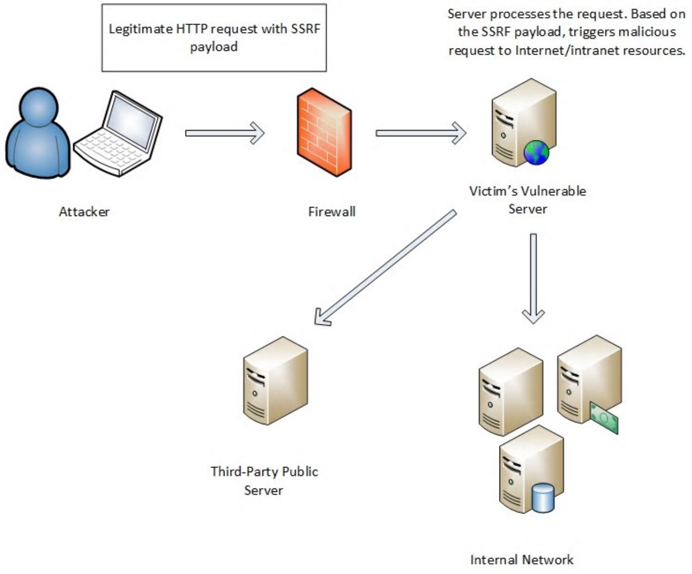
Plain injection	open($input)
Prepended injection	open($input + '/foobar')
Appended injection	open('/foobar' + $input)
Appended and **prepended**	open('/foo'+$input+'/bar)

Exercise

http://basiclfi.challs.cyberchallenge.it/

Server-Side Request Forgery

- A **Server-Side Request Forgery** is a vulnerability that allows an attacker to send a network request from the remote application



Legitimate HTTP request with SSRF payload
Server processes the request. Based on the SSRF payload, triggers malicious request to Internet/intranet resources.
Attacker
Firewall
Victim's Vulnerable Server
Third-Party Public Server
Internal Network

Server-Side Request Forgery

- The impact varies a lot, depending on the control the attacker has on the forged request:
  - Control over the whole **TCP packet**
  - Control over some parts of an **HTTP request**
  - Control only over the **host/port** to which the request is made
  - …

Server-Side Request Forgery

- SSRFs are dangerous because they allow bypassing the firewall
- If the internal network is not properly designed, it is possible to **access to sensible hosts**, like internal web applications and control panels

Server-Side Request Forgery

- If the vulnerable web application is hosted on a **cloud instance**, things become more interesting
- Some instances have access to specials *URLs* that often contain **critical data**

Server-Side Request Forgery

- For example, AWS instances can access the **metadata API**, at the URL http://169.254.169.254/
- This host contains sensible information such as the **IAM security credentials** and general information about the vulnerable instance

Server-Side Request Forgery

- If there is no output, the SSRF is called **blind SSRF**
- It is less dangerous than a normal SSRFs
- With a blind SSRF it is possible to
  - Map the internal network
  - **Trigger actions** on hosts behind the firewall[1]

[1] A nice collection of payloads to use: https://blog.assetnote.io/2021/01/13/blind-ssrf-chains/

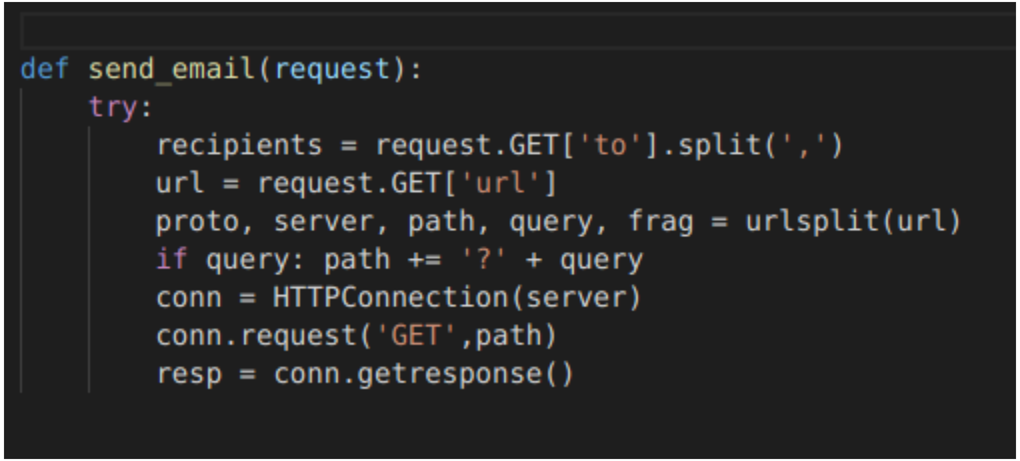
Server-Side Request Forgery

- To find an SSRF, you should:
  - Find suspicious endpoints: If you see a URL inside a parameter try to put a URL controlled by you. You can use a tool like ngrok
  - If you have a pingback at your host, then probably you have an SSRF. Then you should try to insert internal hostnames, like "localhost" or common internal IPs (192.168.1.1,10.0.0.1, and so on..)
  - Examine the response time!

Server-Side Request Forgery

- Every piece of code that can issue a connection can lead to this vulnerability
- Common functions/libraries are:
  - PHP open-like functions
  - CURL
  - Python's urllib
  - ...

Server-Side Request Forgery



```
def send_email(request):
    try:
        recipients = request.GET['to'].split(',')
        url = request.GET['url']
        proto, server, path, query, frag = urlsplit(url)
        if query: path += '?' + query
        conn = HTTPConnection(server)
        conn.request('GET',path)
        resp = conn.getresponse()
```

Server-Side Request Forgery

- Generally speaking, SSRFs are really difficult to avoid
- The most effective way is to check the user-supplied host against a **whitelist**
- Another good mitigation is to make requests from a host that is **isolated from sensitive internal hosts**

Exercise

http://ssrf1.challs.cyberchallenge.it/