Limited File Uploads

So far, we have been mainly dealing with filter bypasses to obtain arbitrary file uploads through a vulnerable web application, which is the main focus of this module at this level. While file upload forms with weak filters can be exploited to upload arbitrary files, some upload forms have secure filters that may not be exploitable with the techniques we discussed. However, even if we are dealing with a limited (i.e., non-arbitrary) file upload form, which only allows us to upload specific file types, we may still be able to perform some attacks on the web application.

Certain file types, like `SVG`, `HTML`, `XML`, and even some image and document files, may allow us to introduce new vulnerabilities to the web application by uploading malicious versions of these files. This is why fuzzing allowed file extensions is an important exercise for any file upload attack. It enables us to explore what attacks may be achievable on the web server. So, let's explore some of these attacks.

XSS

Many file types may allow us to introduce a `Stored XSS` vulnerability to the web application by uploading maliciously crafted versions of them.

The most basic example is when a web application allows us to upload `HTML` files. Although HTML files won't allow us to execute code (e.g., PHP), it would still be possible to implement JavaScript code within them to carry an XSS or CSRF attack on whoever visits the uploaded HTML page. If the target sees a link from a website they trust, and the website is vulnerable to uploading HTML documents, it may be possible to trick them into visiting the link and carry the attack on their machines.

Another example of XSS attacks is web applications that display an image's metadata after its upload. For such web applications, we can include an XSS payload in one of the Metadata parameters that accept raw text, like the `Comment` or `Artist` parameters, as follows:

Limited File Uploads

```
MisaelMacias@htb[/htb]$ exiftool -Comment=' "><img src=1 onerror=alert(window.origin)>' HTB.jpg
MisaelMacias@htb[/htb]$ exiftool HTB.jpg
...SNIP...
Comment                         :  "><img src=1 onerror=alert(window.origin)>
```

We can see that the `Comment` parameter was updated to our XSS payload. When the image's metadata is displayed, the XSS payload should be triggered, and the JavaScript code will be executed to carry the XSS attack. Furthermore, if we change the image's MIME-Type to `text/html`, some web applications may show it as an HTML document instead of an image, in which case the XSS payload would be triggered even if the metadata wasn't directly displayed.

Finally, XSS attacks can also be carried with `SVG` images, along with several other attacks. `Scalable Vector Graphics (SVG)` images are XML-based, and they describe 2D vector graphics, which the browser renders into an image. For this reason, we can modify their XML data to include an XSS payload. For example, we can write the following to `HTB.svg`:

Code: xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN" "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg xmlns="http://www.w3.org/2000/svg" version="1.1" width="1" height="1">
    <rect x="1" y="1" width="1" height="1" fill="green" stroke="black" />
    <script type="text/javascript">alert(window.origin);</script>
</svg>
```

Once we upload the image to the web application, the XSS payload will be triggered whenever the image is displayed.

For more about XSS, you may refer to the Cross-Site Scripting (XSS) module.

Exercise: Try the above attacks with the exercise at the end of this section, and see whether the XSS payload gets triggered and displays the alert.

XXE

Similar attacks can be carried to lead to XXE exploitation. With SVG images, we can also include malicious XML data to leak the source code of the web application, and other internal documents within the server. The following example can be used for an SVG image that leaks the content of (`/etc/passwd`):

Code: xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE svg [ <!ENTITY xxe SYSTEM "file:///etc/passwd"> ]>
<svg>&xxe;</svg>
```

Once the above SVG image is uploaded and viewed, the XML document would get processed, and we should get the info of (`/etc/passwd`) printed on the page or shown in the page source. Similarly, if the web application allows the upload of `XML` documents, then the same payload can carry the same attack when the XML data is displayed on the web application.

While reading systems files like `/etc/passwd` can be very useful for server enumeration, it can have an even more significant benefit for web penetration testing, as it allows us to read the web application's source files. Access to the source code will enable us to find more vulnerabilities to exploit within the web application through Whitebox Penetration Testing. For File Upload exploitation, it may allow us to `locate the upload directory, identify allowed extensions, or find the file naming scheme`, which may become handy for further exploitation.

To use XXE to read source code in PHP web applications, we can use the following payload in our SVG image:

Code: xml

Cheat Sheet

Go to Questions

Table of Contents

Intro to File Upload Attacks

Basic Exploitation

Absent Validation

Upload Exploitation

Bypassing Filters

Client-Side Validation

Blacklist Filters

Whitelist Filters

Type Filters

Other Upload Attacks

Limited File Uploads

Other Upload Attacks

Prevention

Preventing File Upload Vulnerabilities

Skills Assessment

Skills Assessment - File Upload Attacks

My Workstation

OFFLINE

Start Instance

∞ / 1 spawns left

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE svg [ <!ENTITY xxe SYSTEM "php://filter/convert.base64-encode/resource=index.php"> ]>
<svg>&xxe;</svg>
```

Once the SVG image is displayed, we should get the base64 encoded content of `index.php`, which we can decode to read the source code. For more about XXE, you may refer to the Web Attacks module.

Using XML data is not unique to SVG images, as it is also utilized by many types of documents, like `PDF`, `Word Documents`, `PowerPoint Documents`, among many others. All of these documents include XML data within them to specify their format and structure. Suppose a web application used a document viewer that is vulnerable to XXE and allowed uploading any of these documents. In that case, we may also modify their XML data to include the malicious XXE elements, and we would be able to carry a blind XXE attack on the back-end web server.

Another similar attack that is also achievable through these file types is an SSRF attack. We may utilize the XXE vulnerability to enumerate the internally available services or even call private APIs to perform private actions. For more about SSRF, you may refer to the Server-side Attacks module.

DoS

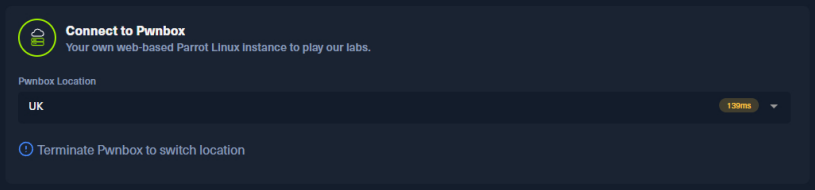
Finally, many file upload vulnerabilities may lead to a `Denial of Service (DOS)` attack on the web server. For example, we can use the previous XXE payloads to achieve DoS attacks, as discussed in the Web Attacks module.

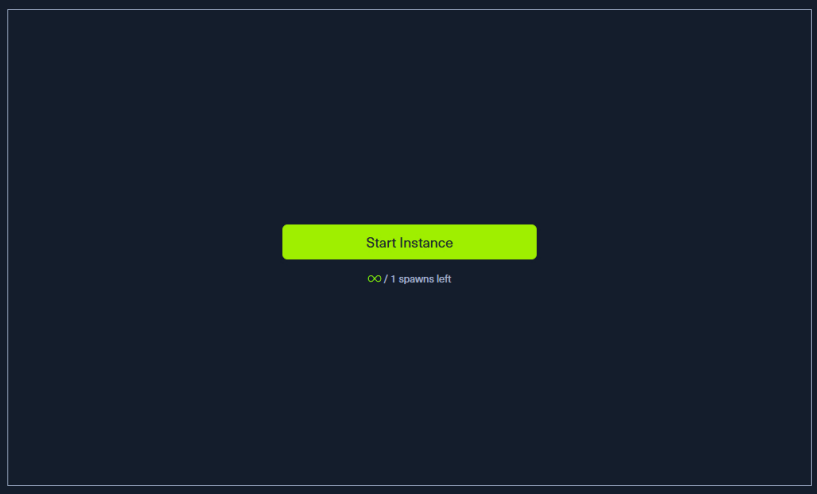
Furthermore, we can utilize a `Decompression Bomb` with file types that use data compression, like `ZIP` archives. If a web application automatically unzips a ZIP archive, it is possible to upload a malicious archive containing nested ZIP archives within it, which can eventually lead to many Petabytes of data, resulting in a crash on the back-end server.

Another possible DoS attack is a `Pixel Flood` attack with some image files that utilize image compression, like `JPG` or `PNG`. We can create any `JPG` image file with any image size (e.g. `500x500`), and then manually modify its compression data to say it has a size of (`0xffff x 0xffff`), which results in an image with a perceived size of 4 Gigapixels. When the web application attempts to display the image, it will attempt to allocate all of its memory to this image, resulting in a crash on the back-end server.

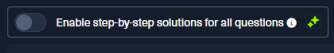
In addition to these attacks, we may try a few other methods to cause a DoS on the back-end server. One way is uploading an overly large file, as some upload forms may not limit the upload file size or check for it before uploading it, which may fill up the server's hard drive and cause it to crash or slow down considerably.

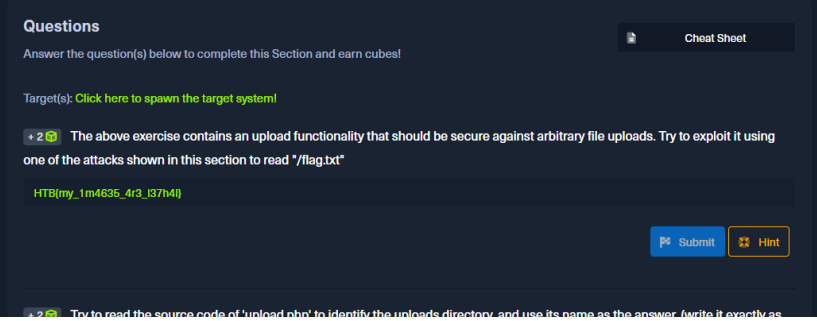
If the upload function is vulnerable to directory traversal, we may also attempt uploading files to a different directory (e.g. `../../../etc/passwd`), which may also cause the server to crash. `Try to search for other examples of DOS attacks through a vulnerable file upload functionality`.





Waiting to start...





+2 Try to read the source code of 'upload.php' to identify the uploads directory, and use its name as the answer. (write it exactly as

found in the source, without quotes)

./images/

Submit Hint

Previous Next

Mark Complete & Next

Powered by HACKTHEBOX