# File Upload Vulnerabilities

**File upload vulnerabilities** occur when a web application allows users to **upload files without proper validation**, leading to **remote code execution**, **malware uploads**, **directory traversal**, or **DoS attacks**.

Such vulnerabilities are common in features like:

* Profile photo uploads
* Document uploads
* File-sharing services
* CMS plugins

# Why It’s Dangerous:

When untrusted users can upload arbitrary files, they may:

* Upload a **web shell** (.php, .jsp, etc.)
* Upload malware disguised as legitimate files
* Perform **path traversal** to overwrite critical files
* Upload large files to **exhaust server resources**

# Impact

| Type | Description |
| --- | --- |
| Remote Code Execution (RCE) | Uploading .php or .jsp file leads to full server control |
| Web Shell Access | Attacker interacts with server via uploaded shell |
| File Overwrite | Replace key config files using path manipulation |
| DoS (Denial of Service) | Upload large/binary files to crash server |
| Stored XSS / Malware | Upload script files that get served to other users |
| Data Exfiltration | Upload backdoor that reads sensitive data |

# Types of File Upload Vulnerabilities

## 1. Unrestricted File Type Upload

* **Definition**: No restriction on file type or extension
* **Example**: Uploading shell.php via profile picture field
* **Mitigation**: Enforce strict extension/content-type allowlist (e.g., jpg, png, pdf only)

## 🔹 2. Content-Type Bypass

* **Definition**: Attacker sets content-type to image/jpeg but uploads .php
* **Example**:

Content-Type: image/jpeg  
Payload: shell.php disguised as image

**Mitigation**: Don’t trust client content-type — verify on server using MIME-type detection

## 3. Double Extension Bypass

* **Definition**: Uploading file like backdoor.php.jpg on systems only checking last extension
* **Mitigation**: Disallow known executable extensions in any part of filename

## 🔹 4. Magic Bytes Bypass

* **Definition**: File extension is .jpg but file content is actually a script
* **Mitigation**: Validate file **magic bytes** (e.g., JPEG starts with FF D8 FF)

## 🔹 5. Directory Traversal / Path Manipulation

* **Definition**: Filename like ../../../../etc/passwd used to escape intended upload directory
* **Mitigation**: Sanitize file paths, use secure upload directories with random names

## 🔹 6. Client-Side Validation Only

* **Definition**: Only JavaScript enforces file type (e.g., .jpg) — attacker bypasses using tools
* **Mitigation**: Enforce validations server-side, not in frontend code

# Testing Techniques

| Technique | Tool | What to Try |
| --- | --- | --- |
| Upload malicious files | Burp Suite, Postman | Try .php, .jsp, .html |
| Check execution path | Burp / Browser | Try accessing file directly via upload path |
| Rename extensions | Local testing | shell.jpg.php, abc.png;.php |
| Analyze response | Observe error or file execution behavior |  |
| Modify content-type | image/jpeg or fake headers |  |
| Use known web shells | Try uploading c99.php, b374k, etc. |  |

# Tools for Testing File Upload Vulns

| Tool | Use |
| --- | --- |
| Burp Suite | Intercept and manipulate upload request |
| Upload Scanner (Burp Ext) | Automated payload testing |
| OWASP ZAP | Passive/active detection |
| LinUpload | Tool to test for common bypass tricks |
| Nikto | May detect exposed upload directories |
| curl / Postman | Send manual crafted file upload requests |

# Mitigation

## 1. File Type & Content Validation

* Allow specific extensions only (.jpg, .png, .pdf)
* Use server-side **MIME type checking** (e.g., image/jpeg)
* Validate **magic bytes**, not just file name or headers

## 2. Rename & Isolate Uploaded Files

* Rename uploaded files to random UUIDs
* Store them **outside web root**
* Serve via a backend handler if needed

## 3. Disallow Executables

* Block extensions: .php, .asp, .aspx, .jsp, .exe, .sh, .bat, .pl, .py
* Sanitize filenames (../, special characters, null bytes)

## 4. Permission Control

* Uploaded files = **non-executable**
* Use chmod 644 or equivalent
* Serve files from static, read-only folder

## 5. Limit Upload Size & Rate

* Prevent DoS by limiting:
  + File size (e.g., 5 MB max)
  + Upload attempts
  + Disk quota per user

## 6. Secure Upload Path

* Don’t allow user control over path
* Avoid appending user input into file paths

## 7. Defense-in-Depth

* Use WAF (Web Application Firewall)
* Antivirus or sandbox scans for uploads
* Implement audit logging for all uploads

# Points

“File upload vulnerabilities are **commonly exploited for RCE**, especially in CMS, file managers, or admin panels.”

“Always enforce **MIME sniffing + magic byte validation + strict allowlists**, and disable execution in upload folders.”

“Check both **storage and execution paths** to ensure that uploads can’t be triggered from browser.”

# Real-World Examples

| Case | Impact |
| --- | --- |
| Joomla / WordPress (many CVEs) | Plugin allowed .php uploads via double extensions |
| ImageTragick (CVE-2016-3714) | Exploited image processing libraries to execute commands |
| Tesla Bug Bounty | Upload bypass in helpdesk portal allowed RCE via Python shell |

# Web Shell PoC (PHP)

<?php system($\_GET['cmd']); ?>  
  
Upload as shell.php, then access: <https://target.com/uploads/shell.php?cmd=whoami>  
That’s a working web shell if server allows execution.