

XSS学习笔记

From <https://www.hackthebox.com/>, Thanks for working

介绍

随着 Web 应用程序变得越来越高级和普遍，Web 应用程序漏洞也越来越多。最常见的 Web 应用程序漏洞类型是跨站点脚本 (XSS) 漏洞。XSS 漏洞利用用户输入中的缺陷将 JavaScript 代码“写入”页面并在客户端执行，从而导致多种类型的攻击。

什么是 XSS

典型的 Web 应用程序通过从后端服务器接收 HTML 代码并将其呈现在客户端互联网浏览器上来工作。当易受攻击的 Web 应用程序未正确清理用户输入时，恶意用户可以在输入字段（例如，评论/回复）中注入额外的 JavaScript 代码，因此一旦其他用户查看同一页面，他们就会在不知不觉中执行恶意 JavaScript 代码。

XSS 漏洞仅在客户端执行，因此不会直接影响后端服务器。它们只能影响执行漏洞的用户。XSS 漏洞对后端服务器的直接影响可能相对较小，但它们在 Web 应用程序中非常常见，因此这相当于中等风险（），我们应该始终通过 `low impact + high probability = medium risk` 检测 `reduce`、修复和修复来尝试冒险。主动防止这些类型的漏洞。



XSS 攻击

XSS 漏洞可以促进范围广泛的攻击，可以是任何可以通过浏览器 JavaScript 代码执行的攻击。XSS 攻击的一个基本示例是让目标用户无意中将他们的会话 cookie 发送到攻击者的 Web 服务器。另一个例子是让目标的浏览器执行导致恶意操作的 API 调用，例如将用户密码更改为攻击者选择的密码。还有许多其他类型的 XSS 攻击，从比特币挖掘到显示广告。

由于 XSS 攻击在浏览器内执行 JavaScript 代码，因此它们仅限于浏览器的 JS 引擎（即 Chrome 中的 V8）。他们无法执行系统范围的 JavaScript 代码来执行系统级代码执行之类的事情。在现代浏览器中，它们也仅限于易受攻击网站的同一域。尽管如此，如上所述，能够在用户的浏览器中执行 JavaScript 仍可能导致各种各样的攻击。除此之外，如果熟练的研究人员发现网络浏览器中的二进制漏洞（例如，Chrome 中的堆溢出），他们可以利用 XSS 漏洞在目标浏览器上执行 JavaScript 漏洞利用，最终突破浏览器的沙箱并在用户的机器上执行代码。

XSS 漏洞可能存在于几乎所有现代 Web 应用程序中，并且在过去二十年中一直被积极利用。一个著名的 XSS 示例是 Samy 蠕虫，这是一种基于浏览器的蠕虫，它在 2005 年利

用了社交网站 MySpace 中存储的 XSS 漏洞。它在查看受感染的网页时执行，方法是在受害者的 MySpace 页面上发布一条消息，内容为“Samy is my hero”。消息本身也包含相同的 JavaScript 负载，以便在其他人的查看时重新发布相同的消息。一天之内，超过一百万的 MySpace 用户在他们的页面上发布了这条消息。尽管这个特定的有效负载没有造成任何实际伤害，但该漏洞可能被用于更邪恶的目的，比如窃取用户的信用卡信息、在他们的浏览器上安装键盘记录器，甚至利用用户网络浏览器中的二进制漏洞（这在当时的网络浏览器中更为常见）。

2014 年，一名安全研究人员意外发现了 Twitter 的 TweetDeck 仪表板中的 XSS 漏洞。此漏洞被利用在 Twitter 中创建一条自我转发的推文，导致该推文在不到两分钟的时间内被转发超过 38,000 次。最终，它迫使 Twitter 在修补漏洞时暂时关闭 TweetDeck。

时至今日，即使是最著名的 Web 应用程序也存在可被利用的 XSS 漏洞。甚至谷歌的搜索引擎页面在其搜索栏中也存在多个 XSS 漏洞，最近一次是在 2019 年，当时在 XML 库中发现了一个 XSS 漏洞。此外，**互联网上最常用的 Web 服务器 Apache Server 曾报告过一个 XSS 漏洞**，该漏洞被积极利用来窃取某些公司的用户密码。所有这些都告诉我们应该认真对待 XSS 漏洞，并且应该付出大量努力来检测和预防它们。

XSS 的类型

XSS 漏洞主要分为三种类型：

类型	描述
Stored (Persistent) XSS 存储型 XSS	最严重的 XSS 类型，当用户输入存储在数据库后端中，然后在检索时显示（例如，帖子或评论）时发生
Reflected (Non-Persistent) XSS 反射型 XSS	当用户输入经过后端服务器处理后显示在页面上，但没有存储（例如，搜索结果或错误消息）时发生
DOM-based XSS DOM 型 XSS	另一种非持久性 XSS 类型，当用户输入直接显示在浏览器中并完全在客户端处理，而不会到达后端服务器（例如，通过客户端 HTTP 参数或锚标记）时发生

存储型 XSS

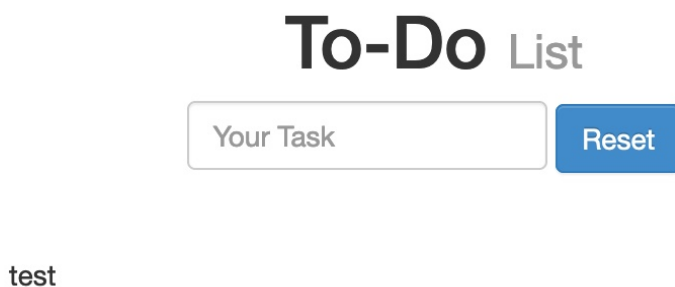
Stored XSS

在我们学习如何发现 XSS 漏洞并利用它们进行各种攻击之前，我们必须首先了解不同类型的 XSS 漏洞及其区别，以便知道在每种攻击中使用哪些。

第一个也是最关键的 XSS 漏洞类型是 **Stored XSS** or **Persistent XSS**。如果我们注入的 XSS 负载存储在后端数据库中并在访问页面时检索，这意味着我们的 XSS 攻击是持久的，并且可能影响访问该页面的任何用户。

这使得这种类型的 XSS 最为严重，因为它会影响更广泛的受众，因为访问该页面的任何用户都将成为这种攻击的受害者。此外，存储型 XSS 可能不容易移除，有效载荷可能需要从后端数据库中移除。

我们可以在下面启动服务器来查看和练习一个 Stored XSS 的例子。如我们所见，网页是一个简单的 **To-Do List** 应用程序，我们可以向其中添加项目。我们可以尝试输入 **test** 并按回车/回车来添加一个新项目，看看页面是如何处理它的：



To-Do List

Your Task

test

如我们所见，我们的输入显示在页面上。如果没有对我们的输入应用清理或过滤，该页面可能容易受到 XSS 攻击。

XSS 测试负载

我们可以使用以下基本 XSS 负载测试页面是否存在 XSS 漏洞：

代码：html

```
<script>alert(window.origin)</script>
```

1. 打开目标Web应用程序并进入需要测试的页面。
2. 将以下代码复制到页面中的任意文本输入框或地址栏中，并点击提交或进入按钮：

```
<script>alert(window.origin)</script>
```

3. 如果页面弹出了当前网页的来源（即协议、主机和端口），则表明该页面存在XSS漏洞。



如我们所见，我们确实收到了警报，这意味着该页面容易受到 XSS 攻击，因为我们的有效负载已成功执行。我们可以通过单击 [**CTRL+U**] 或右键单击并选择 来查看页面源来进一步确认这一点 **View Page Source**，我们应该在页面源中看到我们的有效负载：

代码：html

```
<div></div><ul class="list-unstyled" id="todo"><ul><script>alert(window.origin)</script></ul></ul>
```

hint：许多现代 Web 应用程序使用跨域 IFrame 来处理用户输入，因此即使 Web 表单容易受到 XSS 攻击，它也不会成为主要 Web 应用程序上的漏洞。这就是为什么我们在警告框中显示的值 **window.origin**，而不是像 **. 这样的静态值 1**。在这种情况下，警告框将显示正在执行的 URL，并确认哪个表单是易受攻击的表单，以防使用 IFrame。

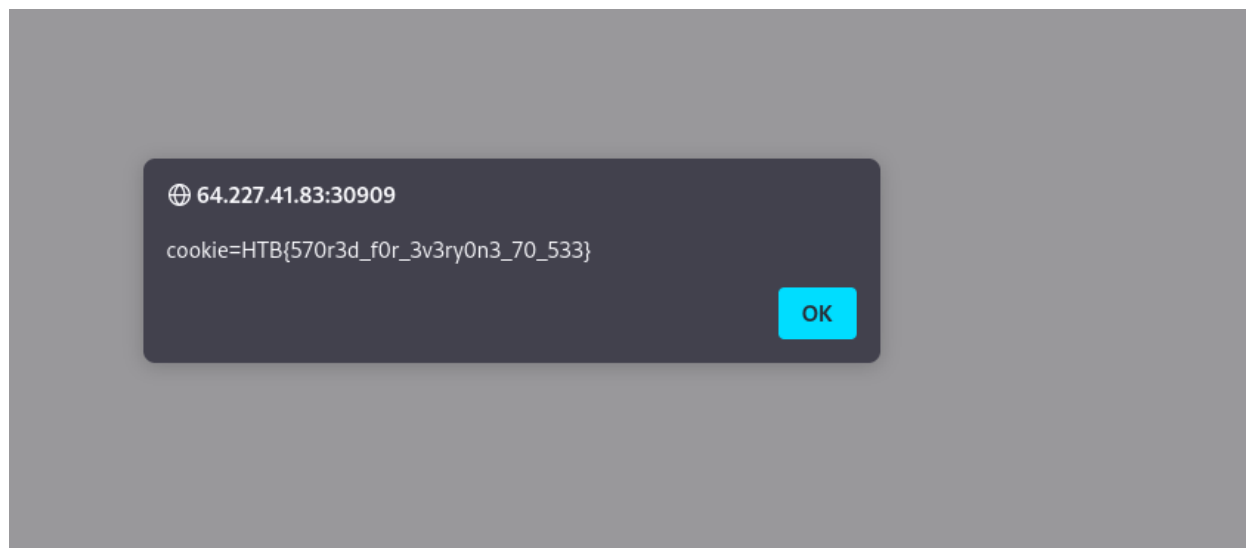
由于一些现代浏览器可能会 `alert()` 在特定位置阻止 JavaScript 功能，因此了解一些其他基本的 XSS 有效载荷可能会很方便，以验证 XSS 的存在。一种这样的 XSS 负载是 `<plaintext>`，它将停止呈现其后的 HTML 代码并将其显示为纯文本。另一个易于发现的有效载荷是 `<script>print()</script>` 会弹出浏览器打印对话框，这不太可能被任何浏览器阻止。尝试使用这些有效载荷来查看每个有效载荷的工作原理。您可以使用重置按钮删除任何当前有效负载。

要查看负载是否持久化并存储在后端，我们可以刷新页面并查看是否再次获得警报。如果我们这样做，我们会看到即使在整个页面刷新过程中我们也会不断收到警报，从而确认这确实是一个 `Stored/Persistent XSS` 漏洞。这对我们来说并不是独一无二的，因为任何访问该页面的用户都会触发 XSS 负载并获得相同的警报。

要获取标志，请使用我们上面使用的相同有效负载，但更改其 JavaScript 代码以显示 cookie 而不是显示 url。

解法：

```
<script>alert(document.cookie)</script>
```



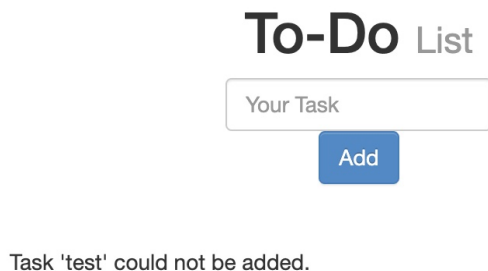
Reflected XSS

反射性XSS

有两种类型的 **Non-Persistent XSS** 漏洞：**Reflected XSS**，由后端服务器处理，和 **DOM-based XSS**，完全在客户端处理，永远不会到达后端服务器。与持久性 XSS 不同，**Non-Persistent XSS** 漏洞是暂时的，不会通过页面刷新持久存在。因此，我们的攻击只会影响目标用户，不会影响访问该页面的其他用户。

Reflected XSS 当我们的输入到达后端服务器并未经过滤或清理就返回给我们时，就会出现漏洞。在许多情况下，我们的整个输入可能会返回给我们，例如错误消息或确认消息。在这些情况下，我们可能会尝试使用 XSS 有效载荷来查看它们是否执行。但是，由于这些通常是临时消息，一旦我们离开页面，它们就不会再次执行，因此它们是 **Non-Persistent**。

我们可以启动下面的服务器在一个易受 Reflected XSS 漏洞的网页上进行练习。**To-Do List** 它与我们在上一节中练习的应用程序类似。我们可以尝试添加任何 **test** 字符串以查看其处理方式：



The screenshot shows a web application titled "To-Do List". It features a text input field labeled "Your Task" and a blue "Add" button below it. Below the button, a red error message is displayed: "Task 'test' could not be added."

如我们所见，我们得到 **Task 'test' could not be added.**，其中包括我们的输入 **test** 作为错误消息的一部分。如果我们的输入没有被过滤或清理，该页面可能容易受到 XSS 攻击。我们可以尝试我们在上一节中使用的相同 XSS 有效载荷，然后单击 **Add**：

```
<script>alert(window.origin)</script>
```

单击后 **Add**，我们会弹出警报：



在本例中，我们看到错误消息现在显示为 `Task '' could not be added.`。由于我们的有效载荷用标签包裹 `<script>`，它不会被浏览器呈现，所以我们得到 `'` 的是空单引号。我们可以再次查看页面源代码以确认错误消息包含我们的 XSS 负载：

```
<div></div><ul class="list-unstyled" id="todo"><div style="padding-left:25px">Task  
'<script>alert(window.origin)</script>' could not be added.</div></ul>
```

正如我们所见，单引号确实包含了我们的 XSS payload `'<script>alert(window.origin)</script>'`

如果我们 `Reflected` 再次访问该页面，错误信息不再出现，我们的 XSS payload 也没有执行，说明这个 XSS 漏洞确实存在 `Non-Persistent`。

但是，如果 XSS 漏洞是非持久性的，我们如何用它来锁定受害者？

这取决于使用哪个 HTTP 请求将我们的输入发送到服务器。`Developer Tools` 我们可以通过单击 `[CTRL+I]` 并选择选项卡来通过 Firefox 进行检查 `Network`。然后，我们可以 `test` 再次放入我们的 payload 并点击 `Add` 发送：

Status	Method	Domain	File
200	GET	localhost	index.php?task=test
200	GET	netdna.bootstrapcdn.com	bootstrap.min.js
200	GET	cdnjs.cloudflare.com	jquery.min.js
404	GET	localhost	favicon.ico

正如我们所看到的，第一行显示我们的请求是一个 GET 请求。GET 请求将它们的参数和数据作为 URL 的一部分发送。所以，to target a user, we can send them a URL containing our payload。要获取 URL，我们可以在发送 XSS 负载后从 Firefox 的 URL 栏中复制 URL，或者我们可以右键单击选项卡 GET 中的请求 Network 并选择 Copy>Copy URL。一旦受害者访问这个 URL，XSS 负载就会执行：



DOM XSS

第三种也是最后一种 XSS 是另一种 Non-Persistent 类型，称为 DOM-based XSS。reflected xss 在通过 HTTP 请求将输入数据发送到后端服务器的同时，DOM XSS 完全通过 JavaScript 在客户端进行处理。当使用 JavaScript 通过 Document Object Model (DOM)。

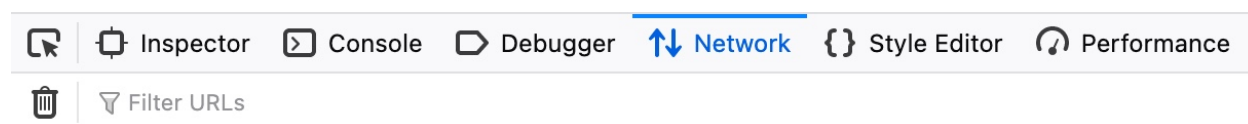
我们可以运行下面的服务器来查看易受 DOM XSS 攻击的 Web 应用程序示例。我们可以尝试添加一个 test item，我们看到 web 应用和我们之前使用的 web 应用类似 To-Do List：



To-Do List

Add

Next Task: test

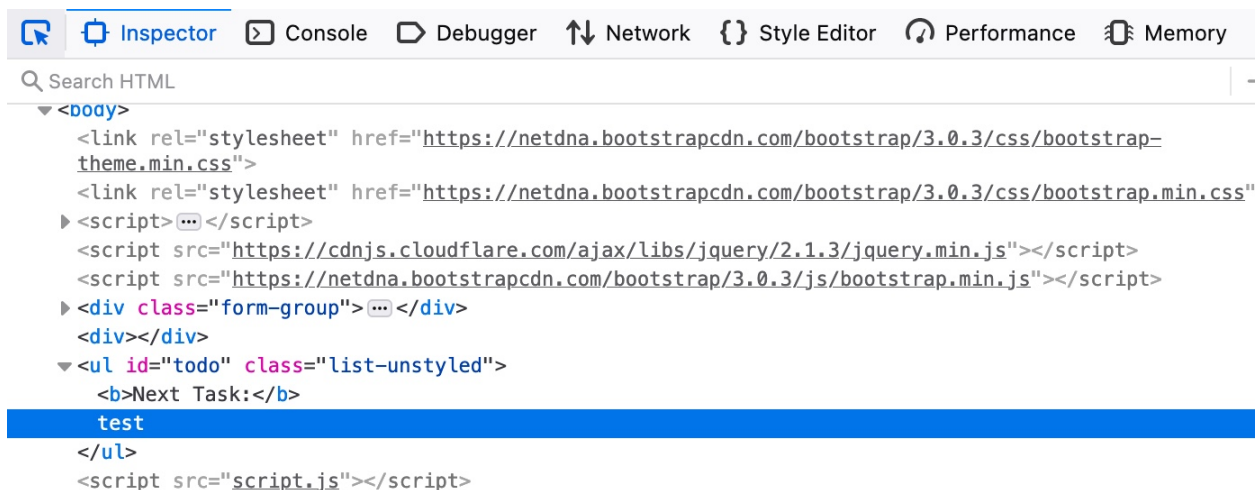
但是，如果我们 **Network** 在 Firefox 开发者工具中打开选项卡并重新添加该 **test** 项目，我们会注意到没有发出 HTTP 请求：



- Perform a request or **Reload** the page to see detailed information about network activity.
- Click on the  button to start performance analysis. 

我们看到 URL 中的输入参数使用了 **#** 我们添加的项目的标签，这意味着这是一个完全在浏览器上处理的客户端参数。这表明输入正在客户端通过 JavaScript 处理，永远不会到达后端；因此它是一个 **DOM-based XSS**。

此外，如果我们通过点击 **[]** 查看页面源代码 **CTRL+I**，我们会注意到我们的 **test** 字符串无处可寻。这是因为当我们点击按钮时，JavaScript 代码正在更新页面 **Add**，这是在我们的浏览器检索到页面源之后，因此基础页面源不会显示我们的输入，如果我们刷新页面，则不会保留（即 **Non-Persistent**）。我们仍然可以通过单击 **[]** 使用 Web Inspector 工具查看呈现的页面源 **CTRL+SHIFT+C**：



```
<body>
  <link rel="stylesheet" href="https://netdna.bootstrapcdn.com/bootstrap/3.0.3/css/bootstrap-theme.min.css">
  <link rel="stylesheet" href="https://netdna.bootstrapcdn.com/bootstrap/3.0.3/css/bootstrap.min.css">
  <script>...</script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/2.1.3/jquery.min.js"></script>
  <script src="https://netdna.bootstrapcdn.com/bootstrap/3.0.3/js/bootstrap.min.js"></script>
  <div class="form-group">...</div>
  <div></div>
  <ul id="todo" class="list-unstyled">
    <b>Next Task:</b>
    test
  </ul>
  <script src="script.js"></script>
```

Source & Sink

要进一步了解DOM-based XSS漏洞的本质，就必须了解页面显示对象的 **Source** 概念。

Sink 是 **Source** 接受用户输入的 JavaScript 对象，它可以是任何输入参数，如 URL 参数或输入字段，如我们上面所见。

另一方面，是 **Sink** 将用户输入写入页面上的 DOM 对象的函数。如果该 **Sink** 函数没有正确清理用户输入，则很容易受到 XSS 攻击。写入 DOM 对象的一些常用 JavaScript 函数是：

- `document.write()`
- `DOM.innerHTML`
- `DOM.outerHTML`

此外，一些 **jQuery** 写入 DOM 对象的库函数是：

- `add()`
- `after()`
- `append()`

如果一个 **Sink** 函数在没有任何清理的情况下写入准确的输入（如上面的函数），并且没有使用其他清理方法，那么我们知道该页面应该容易受到 XSS 攻击。

我们可以查看 To-Do Web 应用程序的源代码，并检查 `script.js`，我们将看到正在 `Source` 从参数中获取 `task=`：

```
1 $(function () {
2   $("#add").click(function () {
3     if ($("#task").val().length > 0) {
4       window.location.href = "#task=" + ($("#task").val());
5       var pos = document.URL.indexOf("task=");
6       var task = document.URL.substring(pos + 5, document.URL.length);
7       document.getElementById("todo").innerHTML = "<b>Next Task:</b> " + decodeURIComponent(task);
8     }
9   });
10 });
11 var pos = document.URL.indexOf("task=");
12 var task = document.URL.substring(pos + 5, document.URL.length);
13 if (pos > 0) {
14   document.getElementById("todo").innerHTML = "<b>Next Task:</b> " + decodeURIComponent(task);
15 }
```

在这些行的正下方，我们看到页面使用该函数在 DOM 中 `innerHTML` 写入变量：`tasktodo`

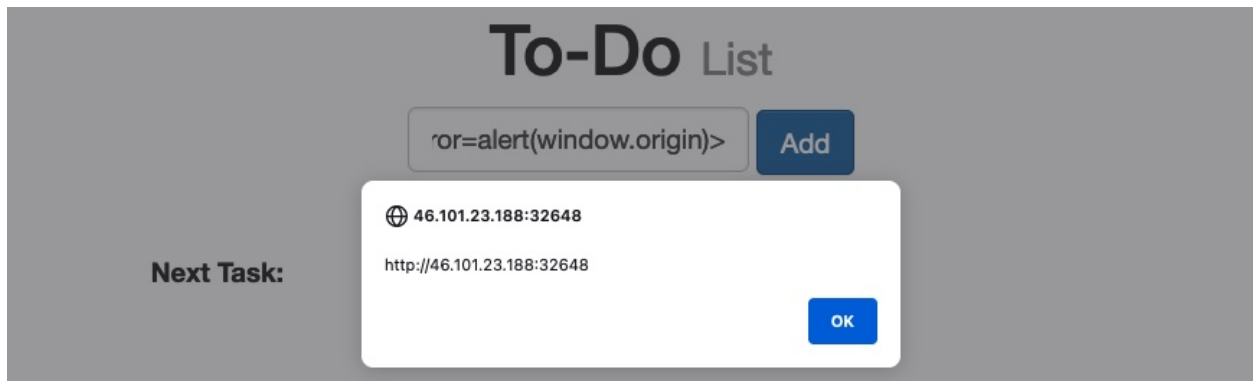
所以，我们可以看到我们可以控制输入，并且输出没有被清理，所以这个页面应该容易受到 DOM XSS 的攻击。

DOM 攻击

如果我们尝试之前使用的 XSS payload，我们会发现它不会执行。这是因为该 `innerHTML` 函数不允许使用 `<script>` 其中的标签作为安全功能。尽管如此，我们使用的许多其他 XSS 有效载荷不包含 `<script>` 标签，例如以下 XSS 有效载荷：

```
<img src="" onerror=alert(window.origin)>
```

上面一行创建了一个新的 HTML 图像对象，它有一个 `onerror` 属性，当找不到图像时可以执行 JavaScript 代码。因此，由于我们提供了一个空图像链接 (`""`)，因此我们的代码应该始终在无需使用标签的情况下执行 `<script>`：



针对具有此 DOM XSS 漏洞的用户，我们可以再次从浏览器复制 URL 并与他们共享，一旦他们访问它，JavaScript 代码就应该执行。这两种有效载荷都属于最基本的 XSS 有效载荷。在许多情况下，我们可能需要根据 Web 应用程序和浏览器的安全性使用各种有效负载，我们将在下一节中讨论。

要获取标志，请使用我们上面使用的相同有效负载，但更改其 JavaScript 代码以显示 cookie 而不是显示 url。

解法：

```
<img src="" onerror=alert(document.cookie)>
```

