5. 客户端安全——请求伪造与欺骗

授课教师：游伟 副教授

授课时间：周五16:00 – 17:30（教二2406）

课程主页：https://www.youwei.site/course/websecurity

目录

5.1 整体概述

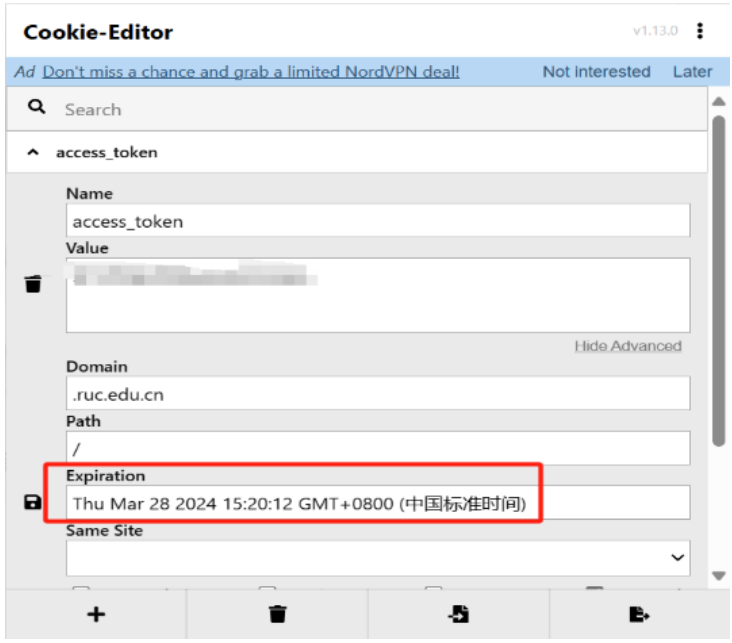
- 前端漏洞总述：

如果说整个**互联网**是一片汪洋大海，那么**网站**就是海洋里星罗棋布的一座座岛屿，而**浏览器**就是一艘船，带着我们在海洋中航行。设想每座岛屿都有一个所属的国家，我们停靠在某座岛屿准备登录的时候，需要亮明**身份**，出示护照和签证，这个签证通常对应HTTP请求头中的Cookie。

5.1 整体概述

- Cookie有效期

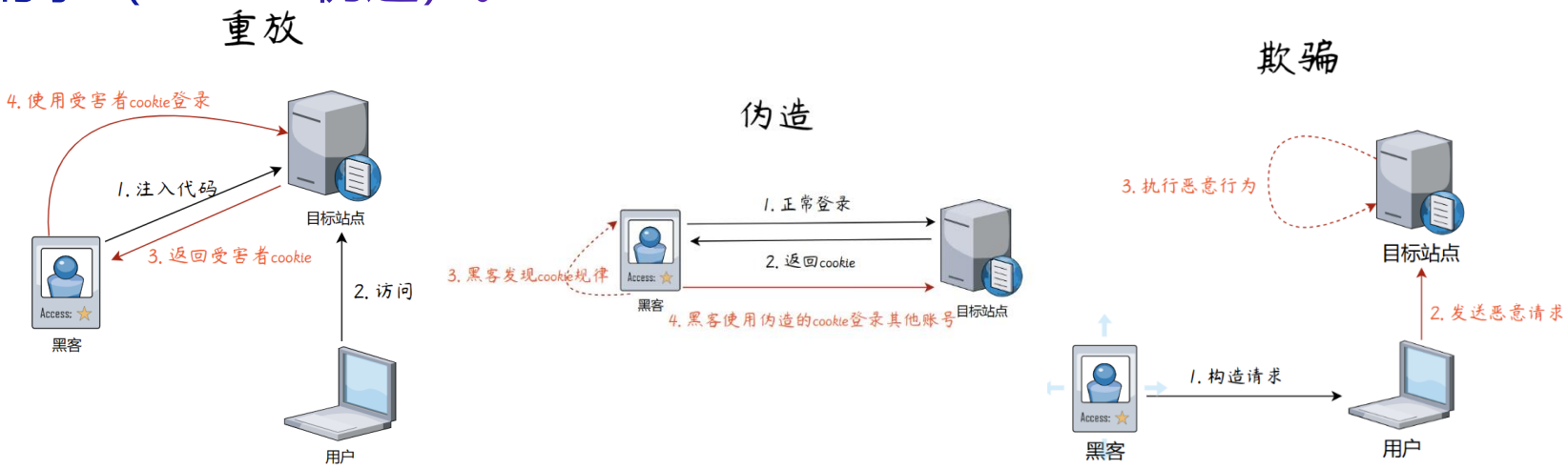
如果我们的签证到期了（Cookie有效期过了），就要重新办理（重新登录）



5.1 整体概述

- Cookie欺骗、伪造、重放

海盗诱导我们去做一些事（钓鱼），这些事造成了很大的危害（cookie欺骗），或者通过某种方式（XSS）获取了我们的签证，用我们的签证去做有害的事（cookie重放）。当然，如果我们的签证加入的验伪信息太少，海盗也能伪造出我们的签证，去做有害的事（cookie伪造）。



5.1 整体概述

本章重点讲述的就是cookie欺骗。

- CSRF

我们在B国购物之后，签证仍未过期，此时我们到访了一个海盗控制的岛屿，海盗给我们一个链接，欺骗我们说点击这个链接就能给我们转账五千万（钓鱼），结果这个链接是让我们在B国的账务给海盗转账（恶意行为）。

CSRF是一个链接导致的恶意行为！

5.1 整体概述

- CORS

你在B国购买东西的时候使用了A国的优惠券，B国会向A国确认你的信息（跨源资源请求），A国返回的信息却向所有人公开（可访问对象设置不正确）。

海盗发现了这个问题，假装了一个店铺（钓鱼），当你在海盗店铺购物的时候，海盗向A国确认你的信息，就能获取A国返回的信息。

CORS是跨源资源的可访问者设置不正确！

5.1 整体概述

- JSON Hijacking

场景与CORS类似，只不过A国返回的结果并不是一段数据，而是一段代码，而代码是默认向所有人公开的。

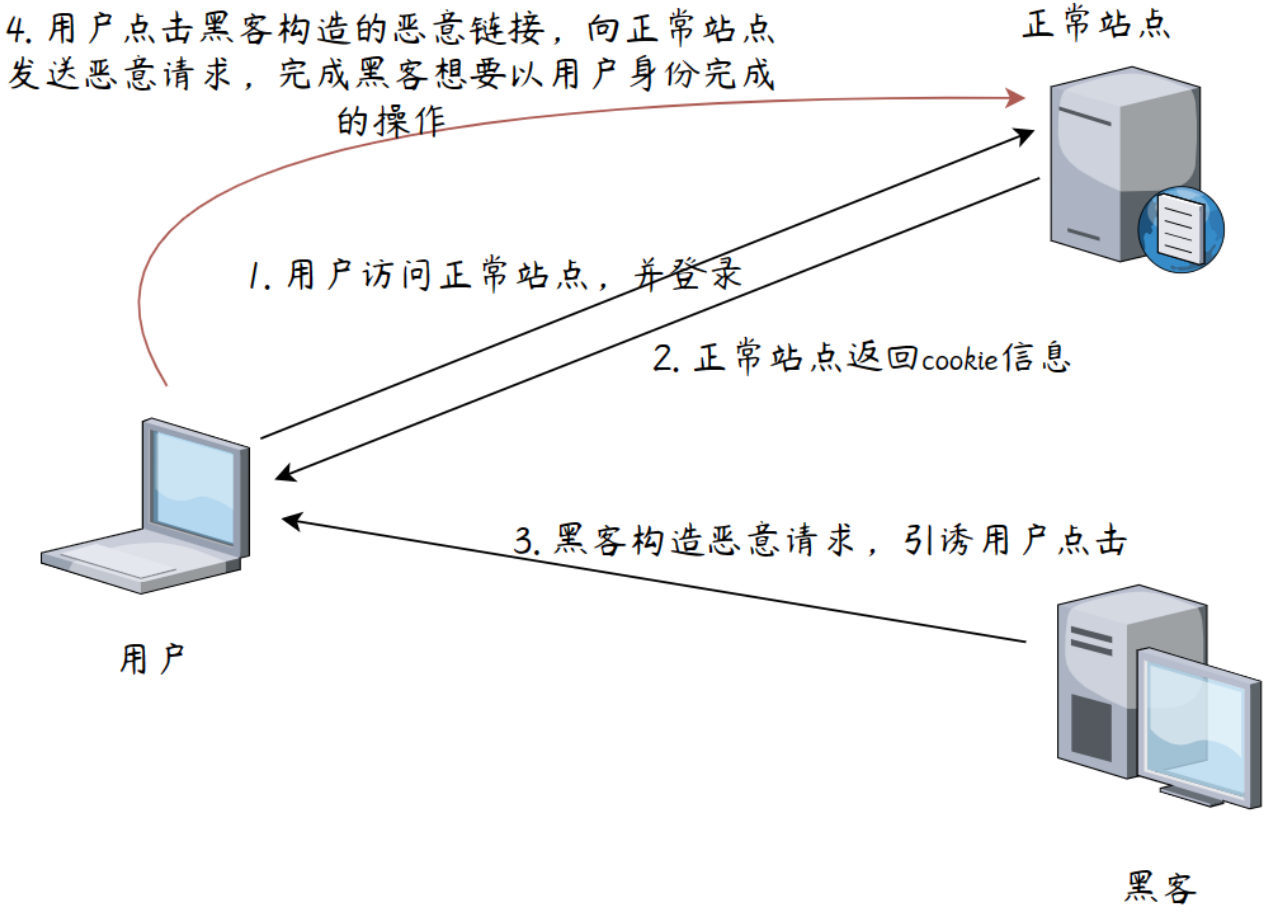
JSON Hijacking是将敏感信息公开！

目录

5.2 CSRF漏洞

- CSRF（Cross-Site Request Forgery），跨站请求伪造攻击，也被称为 one-click attack 或者 session riding。
- CSRF是一种挟制用户在当前已登录的Web应用程序上执行非本意的操作的攻击方法。可以简单的理解为：攻击者可以利用你的登陆信息，以你的身份模拟发送各种请求，对服务器来说这个请求是完全合法的，但是却完成了攻击者所期望的一个操作。
- 比如以你的名义发送邮件、发消息，盗取你的账号，添加系统管理员，甚至于购买商品、虚拟货币转账等。攻击者只要借助少许的社会工程学的诡计，例如通过 QQ 等聊天软件发送的链接（有些还伪装成短域名，用户无法分辨），攻击者就能迫使 Web 应用的用户去执行攻击者预设的操作。
- CSRF一般用于利用用户身份完成某个操作，而非获取数据。

CSRF流程

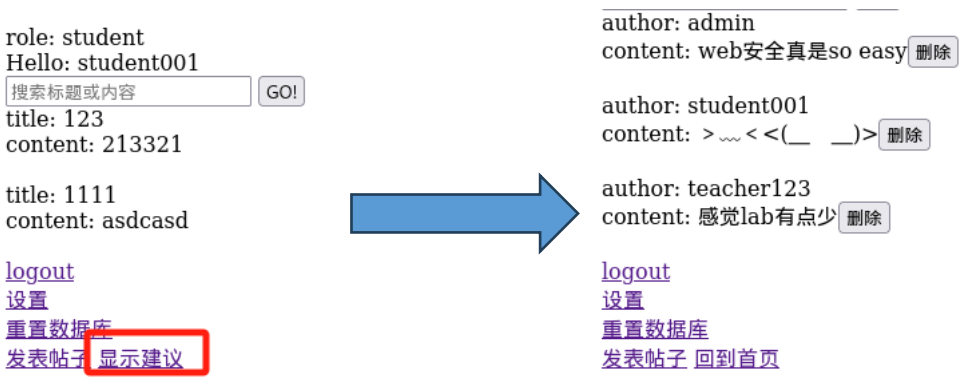

4. 用户点击黑客构造的恶意链接，向正常站点发送恶意请求，完成黑客想要以用户身份完成的操作
正常站点
1. 用户访问正常站点，并登录
2. 正常站点返回cookie信息
3. 黑客构造恶意请求，引诱用户点击
用户
黑客

CSRF原理

- 在上述流程中可以看到，最关键的一步是黑客引诱受害者点击了黑客构造的恶意链接，向正常网站发送的恶意请求，产生这个漏洞的原因是**网站没有检测这个请求是否是正常发出的。**
- 比如说进行转账操作，正常请求的发出应该是在用户进入了网站的转账界面，输入一些东西之后发出的，如果用户在黑客的恶意网站上也发出了同样的操作，应该被视为是危险的，因为这种情况不能确定这个转账操作是不是用户真实想要发出的。
- 从代码上讲，就是**来自不被信任的源的请求也被正常处理。**
- 从社会工程学上讲，**请求能被一步发出**是CSRF漏洞最大的问题，因为这使得受害者没有反应和识别钓鱼网站的机会。

CSRF攻击示例——GET

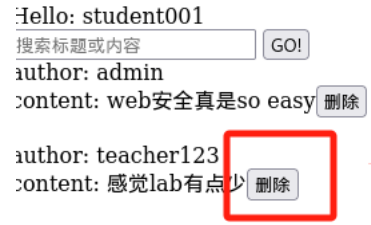
- 通过GET参数传输数据，对攻击者来说利用起来非常方便，因为他们只需要构造一个url诱骗受害者点击就好。
- 打开靶机，登录student001/student001，点击“显示建议”按钮

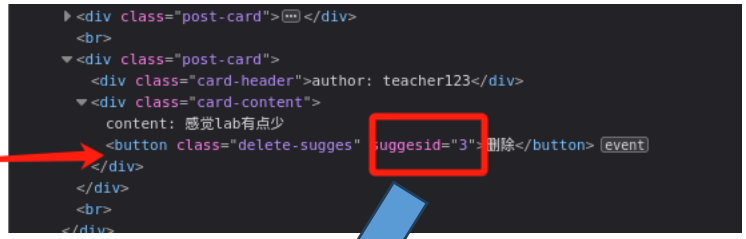


- 一个用户可以点击删除按钮删除自己的帖子，点击别人的删除按钮会遭到拒绝。

CSRF攻击示例——GET

- Student001看到teacher123发出的建议，吓了一跳，想要将其删除。正常操作下，他需要点击页面上的删除按钮。






```
81  function deleteID(id){
82    fetch('delete?id='+id)
83    .then(response =>
84    .then(data => {
85      if (data.status !== 'success
86        alert(data.descrp);
87      } else {
88        location.reload();
89      }
90    })
91    .catch(error => {
92      console.error('Error:', error);
93    });
94  }
```

```
152   <?php if ($_GET['api
153     .then(function()
154           // 删除按
155         document.
156           butto
157             var id = button.getAttribute('suggesID');
158             deleteID(id);
159           });
160         });})
161   <?php } ?>
```

- 点击按钮之后，会发送一个ajax请求，请求地址为http://www.ruc-ctf.site :port/csrf-demo/delete?id=3

CSRF攻击示例——GET

- 很可惜他想删除teacher001的帖子的操作被拒绝了。

- 聪明的student001发现了盲点，他发现删除帖子的id就藏在url中，通过get方式传输

CSRF攻击示例——GET

- 于是他将这个链接http://www.ruc-ctf.site:port/csrf-demo/delete?id=3发送给了teacher001，不过这个链接也太直白了，teacher001一眼就看出这个链接有问题。你能帮帮他怎么让teacher001访问他的钓鱼链接吗?
- 答案是使用**短链接**，可以使用新浪的短链接生成平台：免费短链接生成器 | 即时创建短网址，只有在**公网上有域名**的网址才能生成短链接。
- 由于网站没有认证到底是谁让teacher001访问这个链接（如果是网站自己，那么是正常的，如果不是，那就是异常的），就导致了CSRF漏洞

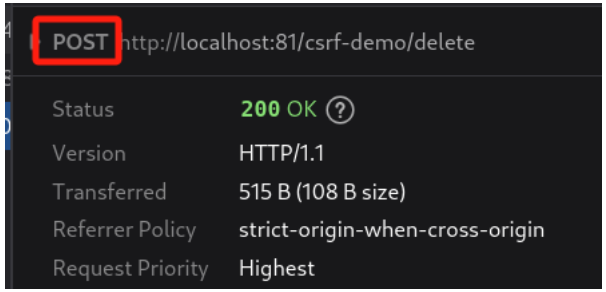
CSRF攻击示例——POST

- Teacher001被骗了之后，让网站赶紧修复了这个漏洞，但是网站开发者很懒，他只是将这个参数换了个传输方式，认为别人控制不了了。
- 思考：post参数真的控制不了吗?
- Student001又访问了vruc，访问设置，修改删除建议方式为POST

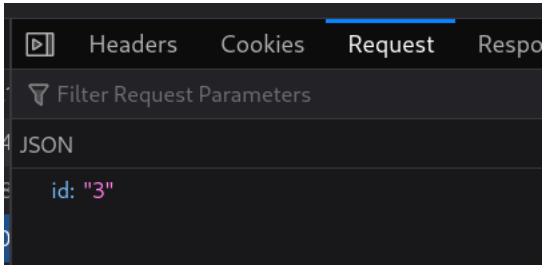

```
83      const formData = new URLSearchParams();
84      formData.append('id', id);
85      fetch('delete', {
86        method: 'POST',
87        headers: {
88        'Content-Type': 'application/x-www-form-urlencoded'
89        },
90        body: formData
91      })
```

CSRF攻击示例——POST

- 他尝试删除一下teacher001的帖子，可以从网络中看到发出的报文



- 查看负载，只通过POST方式传递了一个id值



CSRF攻击示例——POST

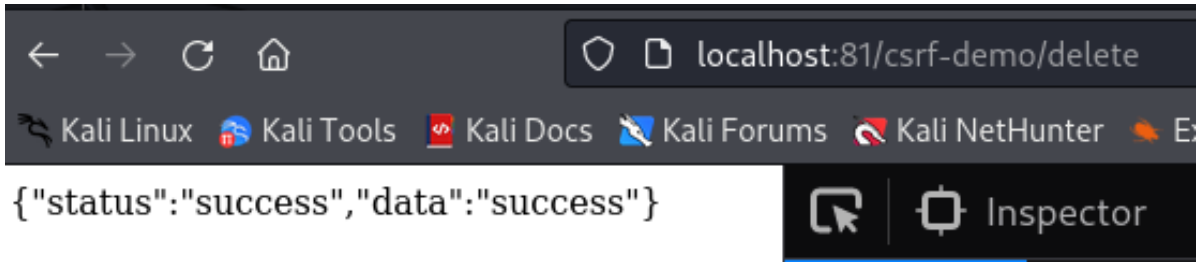
- 聪明的student001想到，我在自己的网站构造这个post请求，让teacher123发出，是不是就能删除teacher123的帖子了。于是他在自己构造的页面中传入了一个名为id的表单值。

```
<!DOCTYPE html>
<html>
    <head>
        <title>post示例</title>
    </head>
    <body>
        <form action="http://ip:port/csrf-demo/delete" method="post" id="form">
            <input name="id" value="3" hidden="hidden">
        </form>
    </body>
    <script>
        setTimeout("document.getElementById('form').submit();",100);
    </script>
</html>
```

将ip:port改为你的靶机

CSRF攻击示例——POST

- Student001需要将构造好的恶意页面放到了自己的网站上，并将这个链接发送给了teacher123，teacher123访问之后就成功的删除了teacher123发送的帖子，造成了CSRF攻击。
- 打开靶机，登录teacher123/teacher，并在相同浏览器中访问https://www.youwei.site/hacker/1.php，输入你的靶机地址(http://www.ruc-ctf.site:28042/csrf-demo/delete)，看到成功删除。（在非实验环境下，可以直接预先定义地址，无需输入）



CSRF的防御

- 根据上面的两个例子，可以总结CSRF的形成原理为以下三条：
  - 攻击者可以预先构造请求
  - 请求能被一步完成，使得受害者没有反应时间
  - 网站没有验证请求是谁让受害者发出的
- 这三个条件缺一不可，早期的安全研究者针对这三点分别提出了三种有效的机制，这三种只需采用任意一种即可：
  - 添加随机Token，使得攻击者无法预先构造请求
  - 多次验证，使得请求不能一步完成
  - 校验用户Referer，验证请求的发出者

防御部分的实验请使用get方式传输

添加随机Token

- 在请求的表单中加入一个参数，这个参数与用户的Session绑定，每个人提交时都不一样，这样，攻击者就无法预测其他用户的参数值，也就无法进行CSRF攻击了。打开靶机，设置防御为token
- 在登录时绑定token

```
136     $_SESSION['name'] = $row['name'];
137     $_SESSION['user_id'] = $row['id'];
138     $_SESSION['role'] = $row['role'];
139     $_SESSION['csrf_token'] = generateNonce();
```

- 在请求的时候参数传递token:

```
84    <?php } else if ($csrf_defense === 1) { ?>
85      fetch('delete?id='+id+"&csrf_token="+"<?= $_SESSION['csrf_token']?>")
86      <?php } ?>
```

- 后端验证token:

```
6   if ($csrf_defense === 1) {
7       if (isset($_GET['csrf_token'])){
8           if ($_GET['csrf_token'] !== $_SESSION['csrf_token']) {
9               throwError("验证不通过");
10          }
11      } else {
12          throwError("缺少csrf_token字段");
13      }
```

添加随机Token

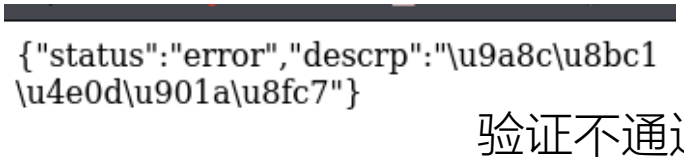
- 实现了token防御后，由于攻击者无法得知受害者的token，无法提前构造恶意url，使得csrf攻击不奏效。
- 访问靶机，登录admin/1234，设置csrf防御为token，提前打开网络，删除admin发送的建议，可以看到建议id为1，后面携带了admin的token

Status	Method	Domain	File
200	GET	localhost:81	delete?id=1&csrf_token=bDRtU3lFRkxWblVhMEd2Qg==

- 重置数据库，访问

http://ip:port/csrf-demo/delete?id=1&csrf_token=123123

可以看到验证不通过



验证不通过

多次验证

- 在提交请求的时候要求用户输入验证码（短信或图形验证码），或者弹出提示框，让用户再次确认是否发出请求，这样能引起用户的警觉，使用户不能仅仅通过点击黑客的恶意链接就完成恶意操作。
- 这种方式并不常用，因为总有受害者不是那么警觉，导致CSRF攻击仍能成功。

多次验证

- 前端设置验证信息

```
100         if (data.status !== 'success') {
101           if (data.descrp === 'need confirm') {
102             if (confirm("你确定要删除这个帖子吗? ")) {
103               fetch('confirm')
104               .then(response => response.json())
105               .then(data => {
106                 if (data.status === 'success') {
107                   deleteID(id);
108                 }
109               })
110             }
111           }
112           else alert(data.descrp);
```

- 后端验证请求，保留验证信息

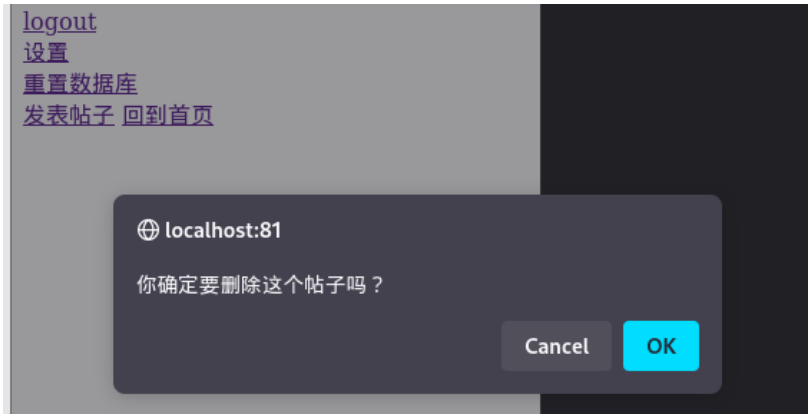
```
1   <?php
2   require_once 'app/utils/doBeforePageStartsWithLogin.php';
3   $_SESSION['confirm'] = 1;
4
5   returnJson("请求已确认");
```

- 执行操作时判断验证信息，**并立即销毁**

```
15      if ($csrf_defense === 2) {
16          if (!isset($_SESSION['confirm'])) {
17              throwError("need confirm");
18          }
19          unset($_SESSION['confirm']);
20      }
```

多次验证

- 访问靶机，登录admin/1234，设置csrf防御方式为多次验证。
- 删除admin发出的建议，发现需要用户点击确认。



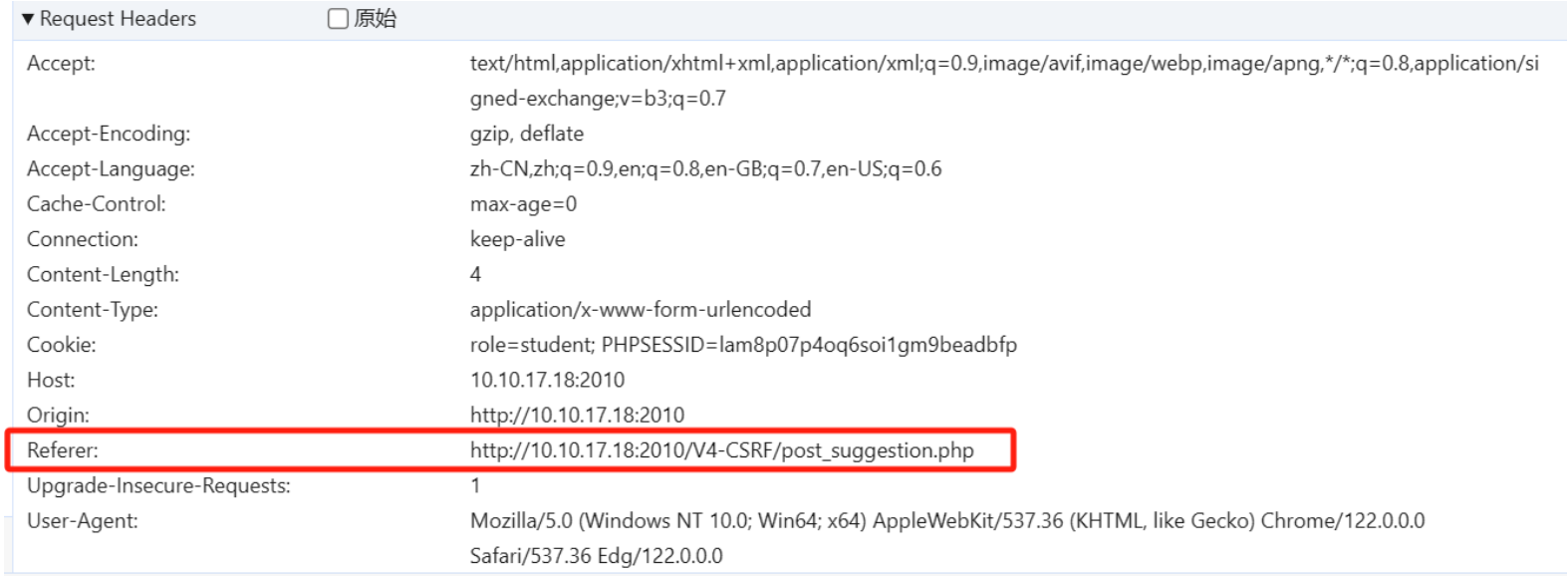
- 重置数据库，直接访问http://ip:port/csrf-demo/delete?id=1，可以看到验证不通过

思考

- 攻击者是否能够通过在一个页面中先发送验证请求，再发送操作请求，来绕过二次验证?
- 不能，由于同源策略的限制，来自攻击者网站的请求会被忽略，攻击者只能通过让用户页面跳转到受害网站来发送请求。使得用户只能访问到验证页面，而不会访问操作页面。

校验用户Referer

- 相比前两种，这种校验referer的思想就比较大胆和冒险了，其设计思想是：由于正常用户提交的请求往往是从本站特定提交的页面发出的，所以该请求会自动带有上一个页面的referer；而攻击者预先构造的请求，当交给用户点击时，往往没有referer或其referer来自攻击者网站或其他网站，所以校验referer也可以进行防御。



▼ Request Headers　☐ 原始

Accept:	text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
Accept-Encoding:	gzip, deflate
Accept-Language:	zh-CN,zh;q=0.9,en;q=0.8,en-GB;q=0.7,en-US;q=0.6
Cache-Control:	max-age=0
Connection:	keep-alive
Content-Length:	4
Content-Type:	application/x-www-form-urlencoded
Cookie:	role=student; PHPSESSID=lam8p07p4oq6soi1gm9beadbfp
Host:	10.10.17.18:2010
Origin:	http://10.10.17.18:2010
Referer:	http://10.10.17.18:2010/V4-CSRF/post_suggestion.php
Upgrade-Insecure-Requests:	1
User-Agent:	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/122.0.0.0 Safari/537.36 Edg/122.0.0.0

校验用户Referer

- 源码：

```
21      if ($csrf_defense === 3) {
22          if (isset($_SERVER['HTTP_REFERER'])) {
23              $scheme = parse_url(url: $_SERVER['HTTP_REFERER'], component: PHP_URL_SCHEME);
24              $host = parse_url(url: $_SERVER['HTTP_REFERER'], component: PHP_URL_HOST);
25              if ($scheme."://".$host !== "http://localhost") {
26                  throwError("referer校验不通过! ");
27              }
28          } else {
29              throwError("referer校验不通过");
30          }
31      }
```

由于靶机限制，我们没有检查端口，这里localhost修改为靶机ip

- 思考：referer不是可以伪造吗，这是怎么防御的？

- 因为CSRF攻击的对象是普通用户，虽然referer的伪造很简单，但难在诱导其他用户去修改Referer（因为这涉及到对网络报文的修改或者主动调用javascript代码），除非网站存在XSS漏洞。

校验用户Referer

- 访问靶机，登录admin/1234，设置csrf防御为referer，将下面url中的ip:port改为你自己的靶机，复制粘贴到浏览器中。
  - http://ip:port/csrf-demo/delete?id=1
- 看到页面识别出CSRF攻击

{"status":"error","descrp":"referer\u6821\u9a8c\u4e0d\u901a\u8fc7"}

Referer校验不通过

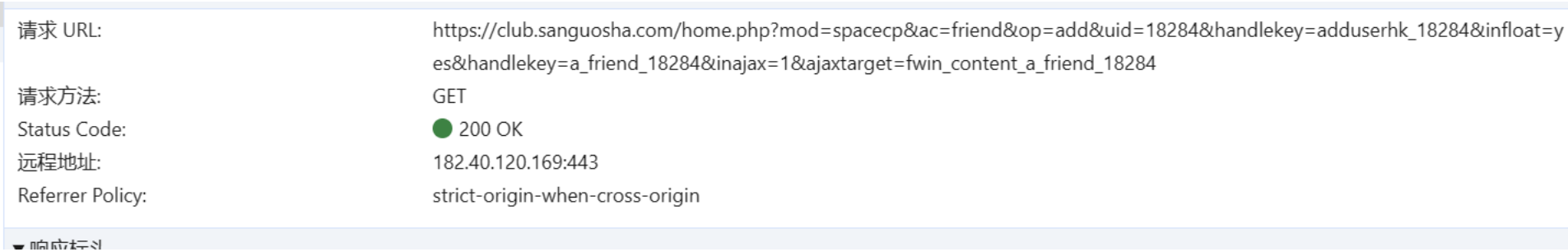
- 因为我们直接这个页面是不带Referer标头的，同样的，从其他源的页面跳转到这个api，referer的校验依然会不通过，因为referer是来自其他源的。

XSS导致的CSRF

- CSRF从原理上能分为两种，一是网站缺乏对操作发出者的检查，另一种是网站存在XSS等漏洞，使得攻击者能以网站为操作发出者发起敏感请求。
- 第一种情况，能够使用前面所讲的三种方法进行防御，第二种情况，即使网站的CSRF防护机制做的再好，也能达成攻击效果。

普通的CSRF

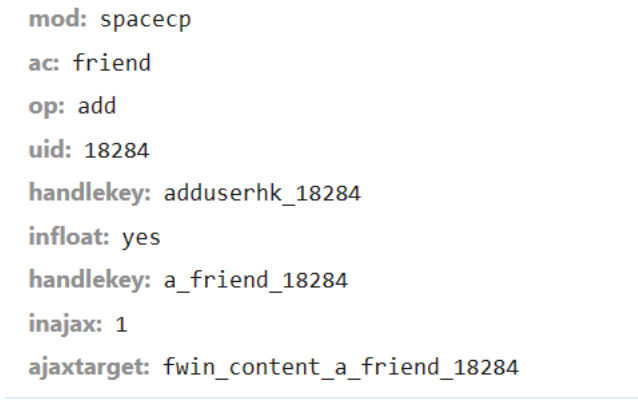
- 登录三国杀网站大家好12341231的好友 - 玩家社区_三国杀官方社区 - Powered by Discuz! (sanguosha.com)
- 在这个页面随便选一个人添加好友，在网络中查看报文



- 查看负载：只有这些get参数

这就是普通的CSRF，直接诱骗

受害者点击这个请求url即可

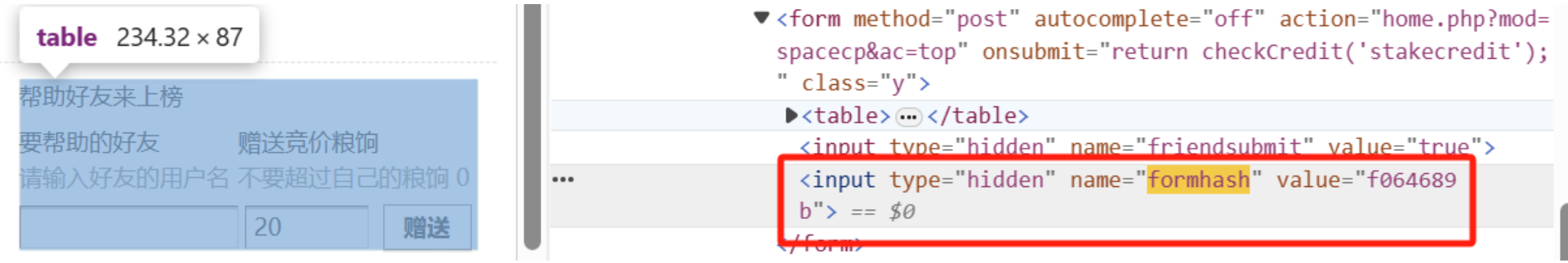


XSS导致的CSRF

- 小明非常喜欢玩三国杀，想让自己的排行上升，但是自己的粮饷太少了，他想通过CSRF的方式让其他用户都给自己转移粮饷。
- 访问用户竞价排行 - - 玩家社区_三国杀官方社区 - Powered by Discuz! (sanguosha.com)，可以看到有一个转移粮饷的位置

XSS导致的CSRF

- 小明满怀期望的看了看这部分的源代码，发现这里居然加了token验证，他顿时泄了气

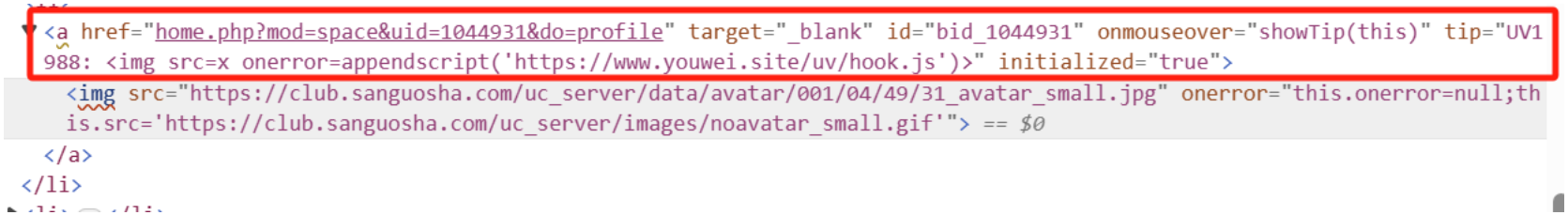


- 此时另一个也喜欢玩三国杀的小红告诉他，三国杀页面存在XSS漏洞，聪明的小明想到可以利用XSS攻击获取别的用户的token值，也就是上面表单的formhash，于是他向小红请教XSS的位置

XSS导致的CSRF

- 小红说访问这个网站玩家社区_三国杀官方社区 – Powered by Discuz! (sanguosha.com)
- 点击我也要上榜后可以在宣言中插入XSS代码

- 由于限制了字数，小明通过以下方式在页面中引入了自己网站的js代码（js代码是可以跨源获取的）



XSS导致的CSRF

- 他通过这段代码获取了用户的token，接下来就为所欲为了！

```
iframe = document.createElement("iframe");
iframe.src = target + "forum.php";
iframe.onload = handler;
//iframe.style = "display:none";
document.body.appendChild(iframe);

function handler() {
  formhash = iframe.contentDocument.getElementsByName("formhash")[0].value;
  alert(formhash);
  my_uid = document.getElementsByClassName("vwmy")[0].firstChild.href.match(/\d+/g)
  alert(my_uid);
```

XSS导致的CSRF

- 思考：这个漏洞怎么修复?

网站虽然限制了输入字数，但是没想到用户能引入跨源js

- 1. 修复XSS注入点
- 2. 页面设置内容安全策略(XSS防御)
- 3. token不要放在前端，而是与session绑定

5.3 JSONHijacking

- JSON 劫持又为“ JSON Hijacking ”，最开始提出这个概念大概是在 2008 年国外有安全研究人员提到这个 JSONP 带来的风险。其实这个问题属于 CSRF攻击范畴。当某网站通过 JSONP 的方式来跨域（一般为子域）传递用户认证后的敏感信息时，攻击者可以构造恶意的 JSONP 调用页面，诱导被攻击者访问来达到截取用户敏感信息的目的。

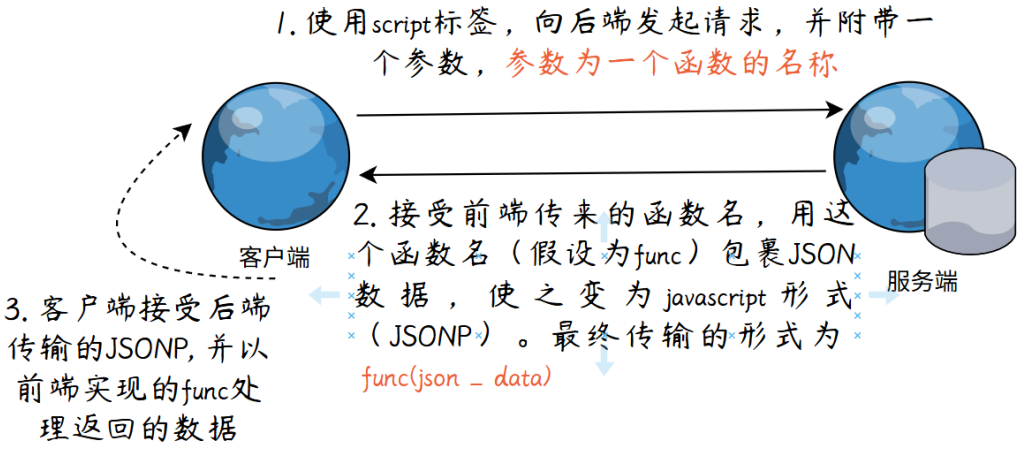
JSON

- JSON 指的是 JavaScript 对象表示法（JavaScript Object Notation）
- JSON 独立于语言：JSON 使用 Javascript语法来描述数据对象，但是 JSON 仍然独立于语言和平台。JSON 解析器和 JSON 库支持许多不同的编程语言。 目前非常多的动态（PHP，JSP，.NET）编程语言都支持JSON。
- JSON从格式上非常接近python的字典和C++的map，都是一个键对应一个值。
- 一个简单的JSON数据实例如图

```
var JSONObject :{name: string, slogan: string, url: string} = {
    "name":"实例",
    "url":"http://www.liuboyu.site",
    "slogan":"JSON真的很简单"
};
```

JSONP

- Jsonp(JSON with Padding) 是 json 的一种"使用模式"，可以让网页从别的域名（网站）那获取资料，即跨域读取数据。
- 为什么我们从不同的域（网站）访问数据需要一个特殊的技术（ JSONP)呢？这是因为同源策略。同源策略禁止了跨源的数据访问，但是没有禁止跨源js代码的请求，JSONP就是将数据变成了代码形式传输，从而绕过同源策略的限制。



JSONP的使用

- 客户端

前面提到，js是可以跨源获取的，客户端需要传递给服务端一个参数，告知服务端 客户端使用哪个函数处理返回的数据

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>JSONP实例</title>
    </head>
    <body>
        <div id="result-container">
        </div>
        <script>
            function callbackFunc(result, methodName) {
                document.getElementById("result-containner").innerHTML = result;
            }
        </script>
        <script src="http://www.ruc-ctf.site/jsonp.php?jsoncallback=callbackFunc"></script>
    </body>
</html>
```

JSONP的使用

- 服务端

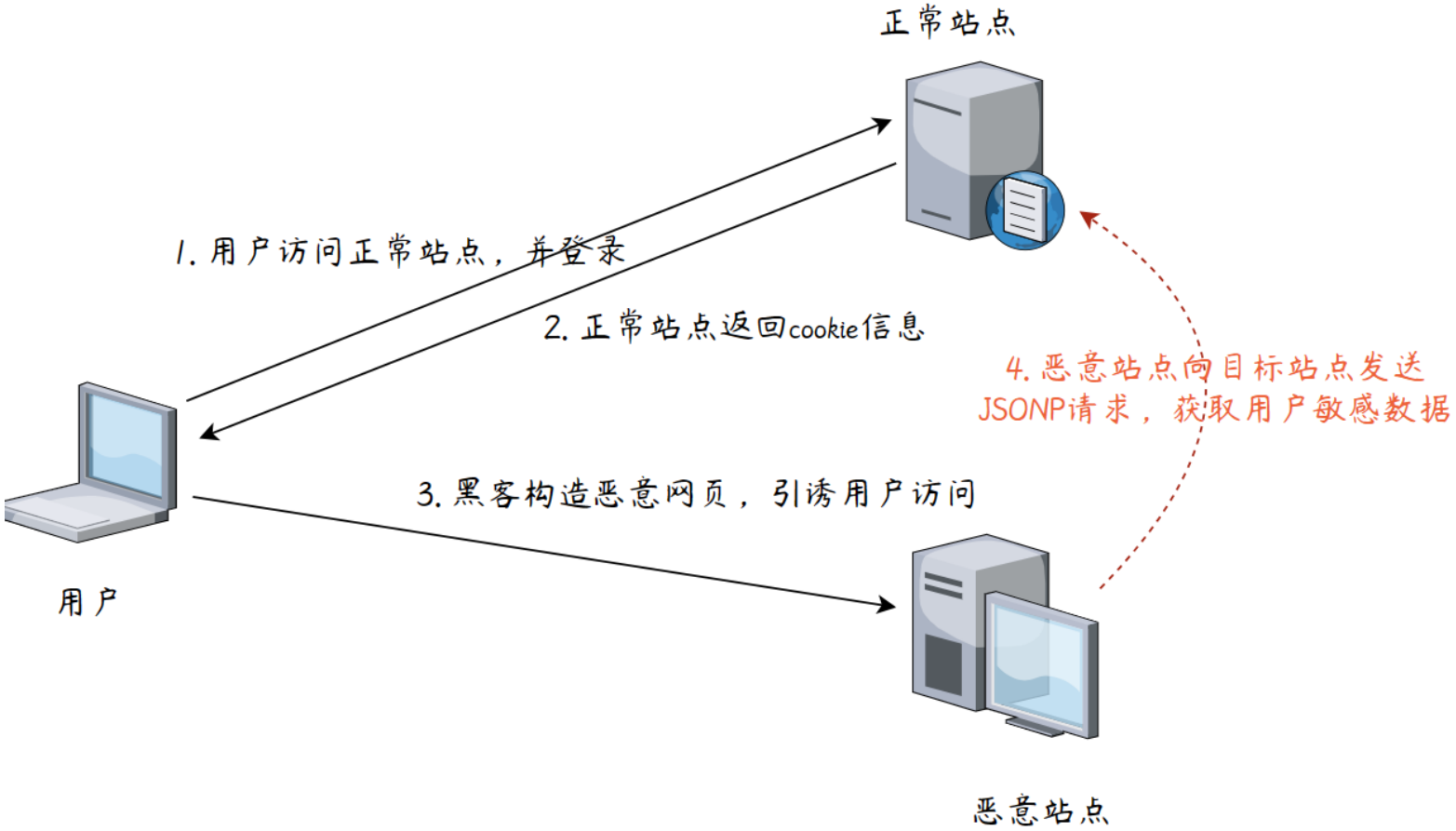
接受客户端传来的参数，用这个函数名包裹JSON形式的数据，这样，就能以js代码的形式返回数据，“欺骗”同源策略，使得前端能够获取跨源的数据。

```php
<?php
header( header: 'Content-type: application/json');
//获取回调函数名
$jsoncallback = htmlspecialchars($_REQUEST ['jsoncallback']);
//json数据
$json_data = '["customername1","customername2"]';
//输出jsonp格式的数据
echo $jsoncallback . "(" . $json_data . ")";
?>
```

JSON劫持原理

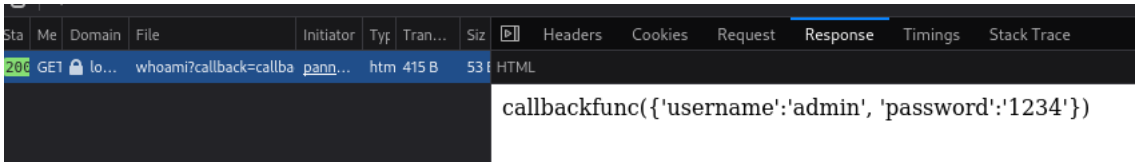
- 当网站以JSONP形式传输数据，且其中包含用户敏感信息，攻击者通过第三方站点以CSRF手段使用户浏览器请求目标站点得到包含敏感信息的JSON数据，进而劫持到敏感信息。
- Json劫持攻击过程有点类似于csrf，只不过csrf的目的是以用户身份完成操作，但是json-hijack的目的是获取敏感数据。
- 原理：一些web应用会把一些敏感数据以jsonp的形式返回到前端，如果仅仅通过cookie来判断请求是否合法，那么就可以利用类似csrf的手段，向目标服务器发送请求，以获得敏感数据。

JSON劫持流程


正常站点
1. 用户访问正常站点，并登录
2. 正常站点返回cookie信息
4. 恶意站点向目标站点发送
JSONP请求，获取用户敏感数据
3. 黑客构造恶意网页，引诱用户访问
用户
恶意站点

JSON劫持实例

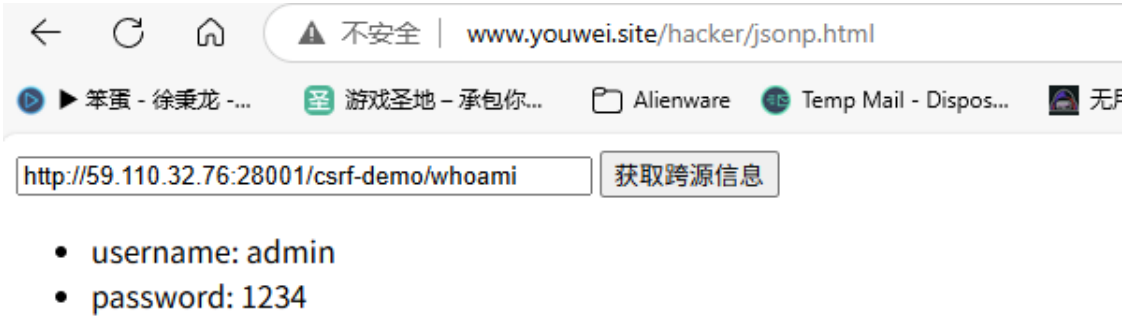
- 访问靶机，登录admin/1234，点击“我是谁”按钮，可以看到用户信息被输出到页面上，通过查看页面源代码或者网络的方式发现此处使用了jsonp方式传输数据。




```
79      <button onclick="getwhoami()">我是谁? </button>
80    </div>
81  </body>
82  <script nonce="<?= $nonce ?>">
83    function getwhoami(){
84      var script = document.createElement('script');
85      script.src = 'whoami?callback=callbackfunc';
86      document.body.appendChild(script);
87    }
88
89    function callbackfunc(res, funcname) {
90      var html = "<ul>";
91      for (x in res) {
92        html += '<li>' + res[x] + '</li>';
93      }
94      html += '</ul>';
95      document.querySelector(".whoami-container").innerHTML = html;
96    }
```

JSON劫持实例

- 由于js的跨域共享机制，这段代码可以同样可以被恶意服务器获取。访问http://www.youwei.site/hacker/jsonp.html，输入你的靶机地址http://59.110.32.76:28001/，即可看到攻击效果。



JSON劫持防御

- 因为JSON劫持是特殊的一种CSRF攻击，所以所有CSRF防御手段对JSON劫持防御都有效
- 验证码，token，referer

5.4 CORS

- Cors全称"跨域资源共享"（Cross-origin resource sharing），Cors的出现是用来弥补SOP（同源策略）的不足。在当时SOP有些限制了网页的业务需求，不能够使不同域的网页互相访问，因此提出了Cors：用于绕过SOP（同源策略）来实现跨域资源访问的一种技术。
- Cors漏洞就是攻击者利用Cors技术来获取用户的敏感数据，从而导致用户敏感信息泄露。
- CORS攻击的目的是获取敏感数据

CORS的设置

- 服务器可以设置自己的资源能被谁访问、被通过哪种方式访问，一个简单的示例如下

```
// 允许来自所有域的跨域请求
header( header: "Access-Control-Allow-Origin: *");
// 允许的请求方法
header( header: "Access-Control-Allow-Methods: GET, POST, PUT, DELETE, OPTIONS");
// 允许的请求头
header( header: "Access-Control-Allow-Headers: Content-Type, Authorization");
// 是否允许发送Cookie
header( header: "Access-Control-Allow-Credentials: true");
```

- 其中最重要的，与同源策略相关的，就是Access-Control-Allow-Origin，这里设置为*，就是所有域都能访问，也就是说，不管是谁请求了这个接口，都能获取返回数据。

CORS的设置

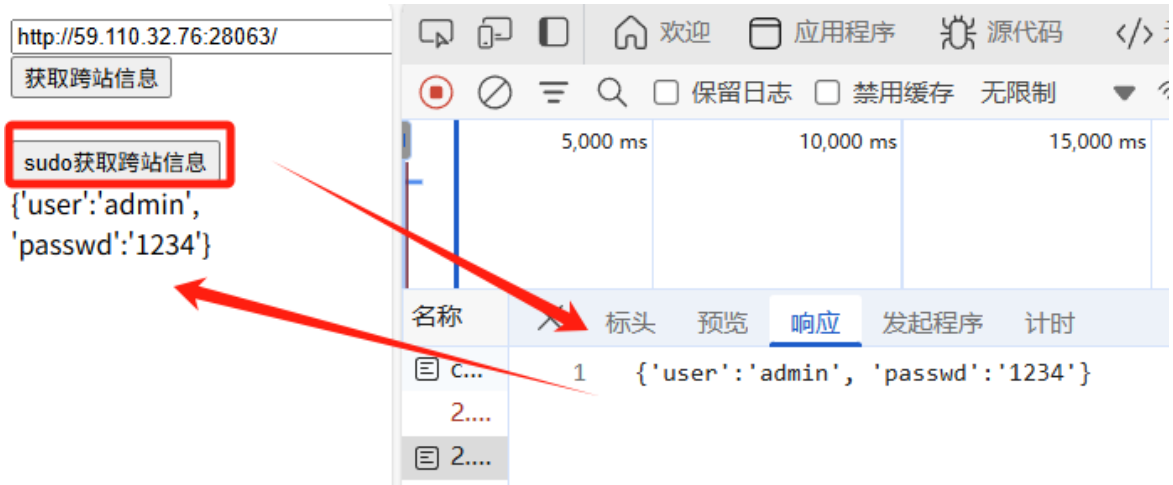
- 如果没有设置前面所说的Access-Control-Allow-Origin，请求会被阻止，打开靶机，登录admin/1234。访问http://www.youwei.site/hacker/cors.html，输入你的靶机地址http://59.110.32.76:28063/，打开控制台，点击按钮“获取跨站资源信息”，可以看到跨源资源请求被阻止：

CORS的设置

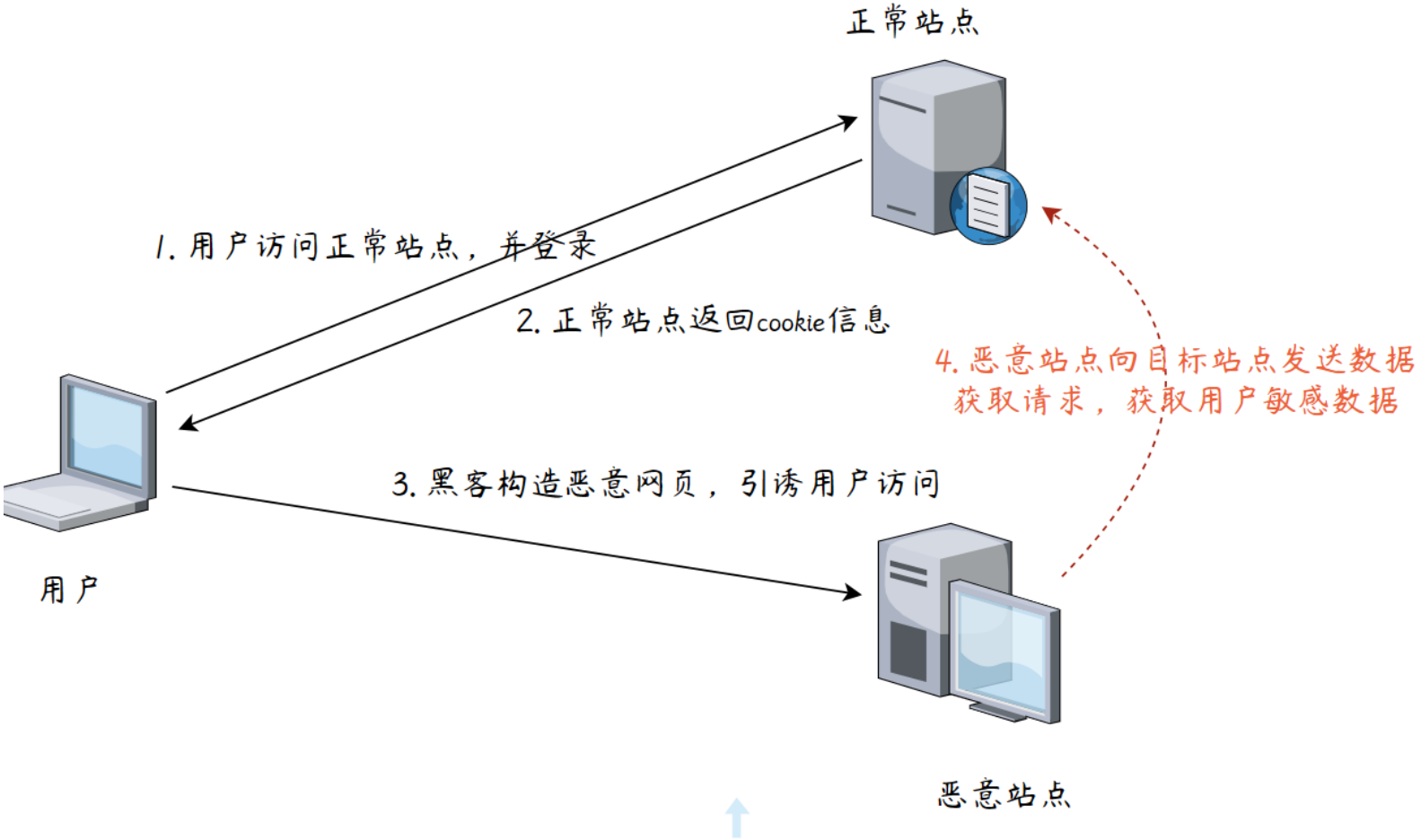
- 如果设置了Access-Control-Allow-Origin，允许全部源访问：

```php
<?php
header(header: "Access-Control-Allow-Origin: *");
echo "{'user':'admin', 'passwd':'1234'}";
```

- 资源就能被该源访问，访问靶机，登录admin/1234，点击 "sudo获取跨源信息"，就能看到数据被输出到页面上。



CORS攻击流程


正常站点
1. 用户访问正常站点，并登录
2. 正常站点返回cookie信息
4. 恶意站点向目标站点发送数据
获取请求，获取用户敏感数据
3. 黑客构造恶意网页，引诱用户访问
用户
恶意站点

CORS攻击

- 攻击原理即为网站将敏感数据的请求接口的Access-Control-Allow-Origin设置为*
- 与CSRF相同，攻击者只需要构造一个请求该数据接口的页面，然后诱骗受害者点击，如果受害者登录了目标站点，攻击者就能成功的获取受害者的信息。
- 可以通过靶机中的“sudo获取跨源信息”体验攻击效果

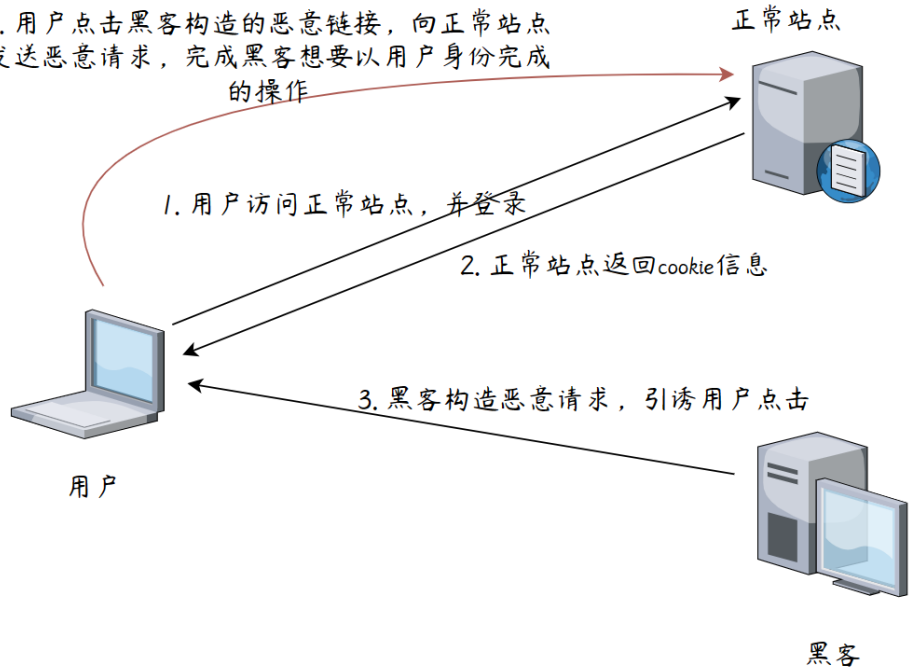
CORS防御

- 不要将敏感数据的请求接口的Access-Control-Allow-Origin设置为*
- 任何CSRF防御措施都能防御CORS

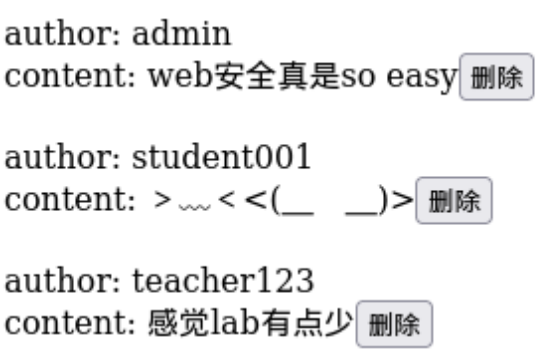
5.5 Clickjacking

- 书接上文，teacher001的帖子被student001通过CSRF漏洞删除，之后他的信息又通过JSONHijacking和CORS漏洞被student001获取，这导致他非常生气，让vruc开发者对这几个漏洞进行了非常完备的修复。

CSRF攻击流程



- Student001尝试了多次后，发现网站确实不存在CSRF漏洞了，但是teacher001的评论令他十分难受



- 聪明的student001想了想，既然不能模拟受害者发出恶意报文或者访问恶意url（CSRF），那能不能在teacher123不知情的情况下让他真的以“**正常**”操作的方式删除评论呢?

5.5 ClickJacking

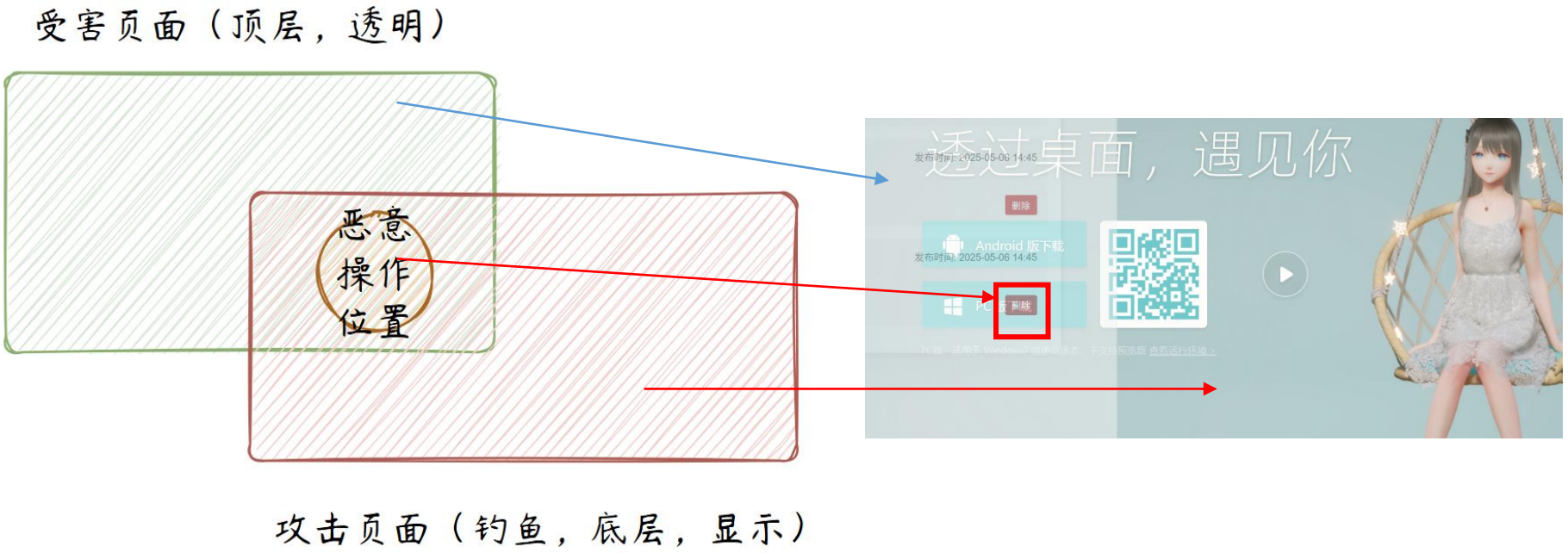
- 点击劫持（Clickjacking）攻击，又称为界面伪装攻击，是一种利用**视觉欺骗手段**进行攻击的方式。
- 攻击者通过技术手段欺骗用户点击本没有打算点击的位置，当用户在被攻击者攻击的页面上进行操作时，实际点击结果被劫持，从而被攻击者利用。
- 这种攻击方式利用了用户对网站的信任，通过覆盖层（通常是透明的iframe）覆盖在另一个网页之上，使受害者无法察觉。
- 点击劫持的目的是让受害者在不知情的情况下执行恶意操作。

点击劫持的原理

- 点击劫持是视觉欺骗，用户只看到了底层的钓鱼页面，与页面进行交互时却是与上层的攻击页面在交互。这是由于透明的iframe造成的，通过控制iframe的位置，导致上层页面的按钮等覆盖到下层上。

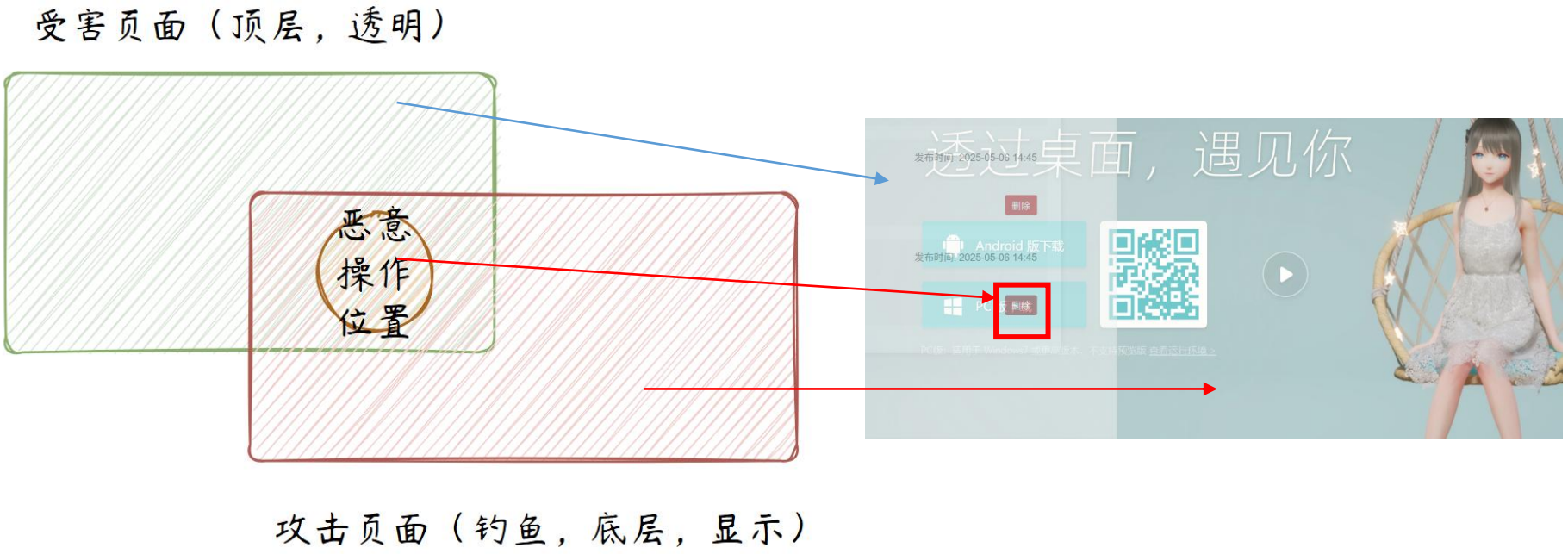
点击劫持攻击的三个关键点

- 视觉欺骗：攻击者使用一个透明的 iframe 覆盖在目标网页上，由于 iframe 是透明的，用户看不到顶层的实际内容，从而在执行操作时，以为是点击了底层的内容，但实际上是点击了顶层 iframe 中的内容。



点击劫持攻击的三个关键点

- 视觉伪装：为了进一步迷惑用户，攻击者通常会在其创建的网页上放置一些吸引用户的元素，如游戏、视频播放器等，当用户点击这些看似无害的区域时，实际上触发的是隐藏在其下的目标网站中的敏感操作。

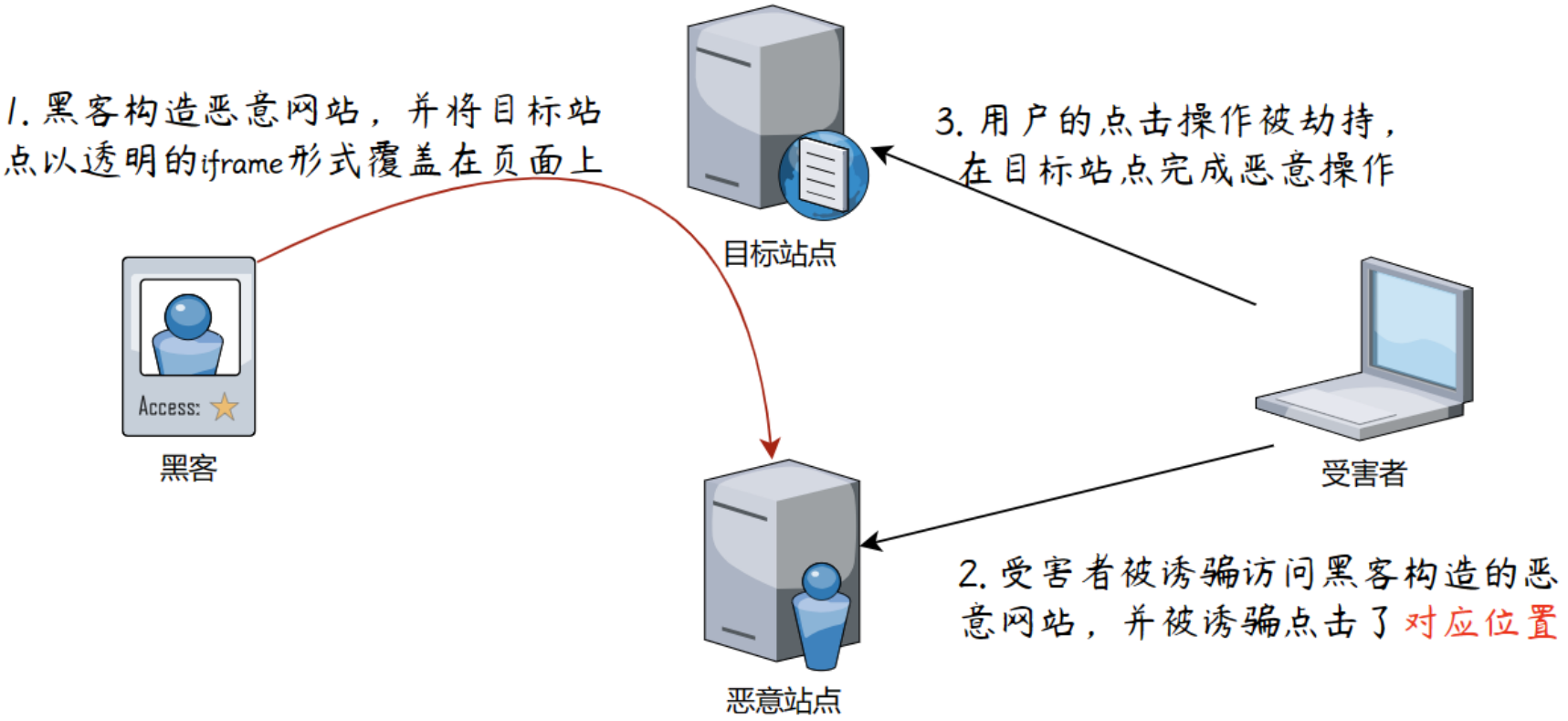


点击劫持攻击的三个关键点

- 用户交互：当用户尝试点击覆盖层上的按钮或链接时，实际上他们点击的是下面的 iframe 中目标网站上的按钮或链接。然后会执行相应的恶意操作，如提交表单、跳转链接等，从而达到攻击者的目的。

- 点击劫持攻击与 CSRF 攻击有异曲同工之妙，都是在用户不知情的情况下诱使用户完成一些动作。但是在 CSRF 攻击的过程中，如果出现用户交互的页面，则攻击可能会无法顺利完成。与之相反的是，点击劫持没有这个顾虑，它**利用的就是与用户产生交互的页面**。

点击劫持攻击流程


1. 黑客构造恶意网站，并将目标站点以透明的iframe形式覆盖在页面上
目标站点
3. 用户的点击操作被劫持，在目标站点完成恶意操作
Access:
黑客
受害者
2. 受害者被诱骗访问黑客构造的恶意网站，并被诱骗点击了对应位置
恶意站点

点击劫持攻击示例

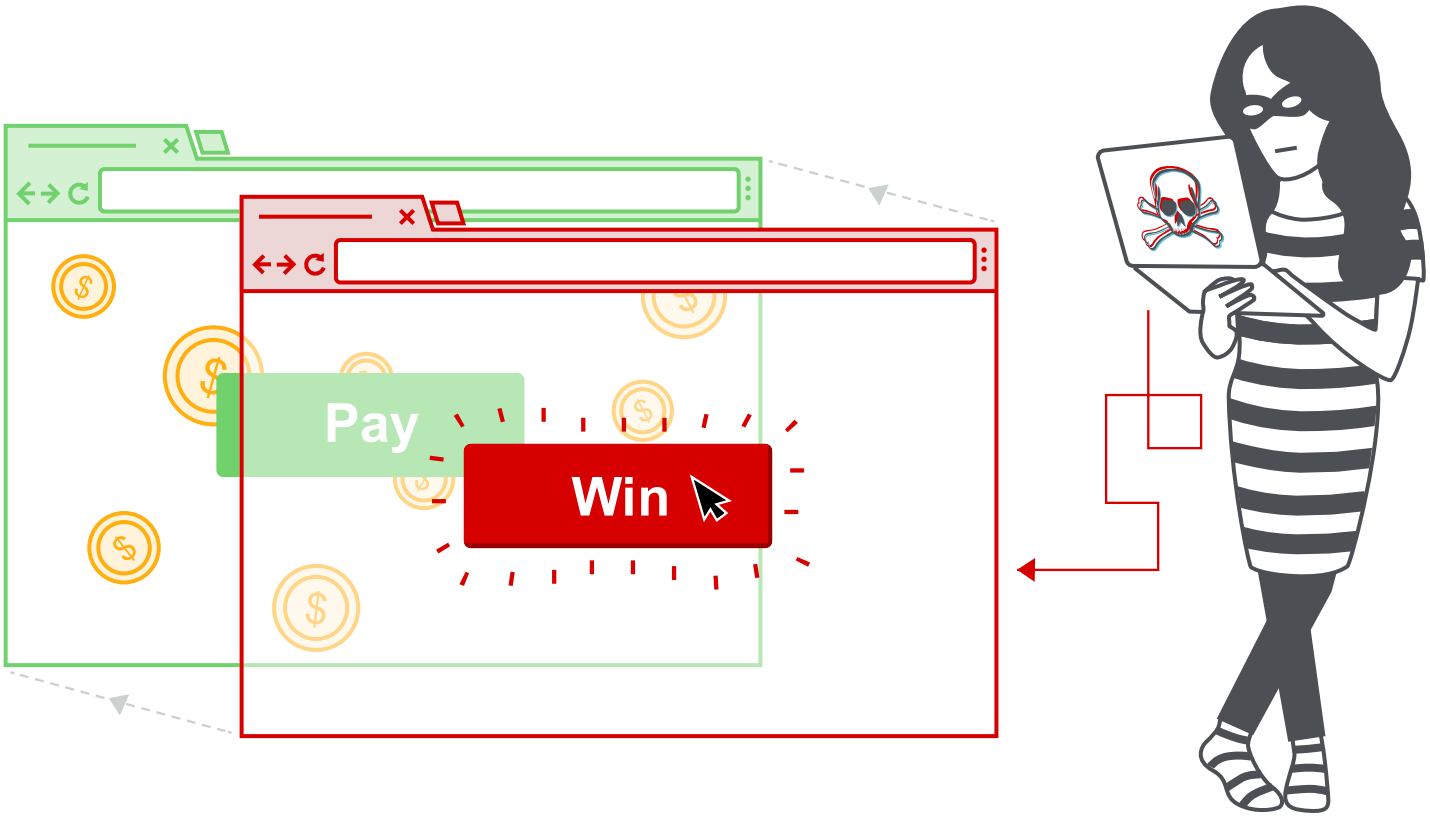
- 访问 https://victim.ruc-ctf.site/click-jacking/dashboard.php ，选择无防御，并登录teacher/teacher。
- 访问钓鱼链接https://hacker.ruc-ctf.site/click-jacking/，可以看到teacher的帖子删除按钮覆盖在pc下载按钮上面。

- 点击按钮，teacher帖子成功删除，你可以点击左上角的按钮隐藏受害页面，达成更好的攻击效果。

点击劫持的类型

- Iframe覆盖攻击

前面讲的主要就是iframe覆盖攻击



点击劫持的类型

- Flash点击劫持

比起浏览页面，与用户互动性很高的Flash游戏是Clickjacking攻击的重灾区。攻击者通过设计一些需要用户点击的小游戏，来诱使用户完成一组操作，完成一些不可告人的目的，比如打开用户的摄像头。所以，在一些小网站上的游戏最好不要玩，如果一定要玩，也要确保之间没有打开网上银行等重要网站，最好可以关闭浏览器的扩展与插件。

点击劫持的类型

- 图片覆盖攻击（XSS攻击与点击劫持攻击结合）

一名叫 sven.vetsch 的安全研究者最先提出了这种 Cross Site Image Overlaying 攻击，简称XSIO。这种攻击的原理是利用网站的XSS漏洞向其中插入恶意标签和图片

如果覆盖一些用户经常点击的位置，比如**网站logo，人物头像，按钮**等，如果用户此时再去点击 这些文职，则会被链接到受害者构建的恶意网站（钓鱼网站或者CSRF链接），或者执行恶意操作

访问 https://victim.ruc-ctf.site/click-jacking/xss.php ，登录 teacher/teacher

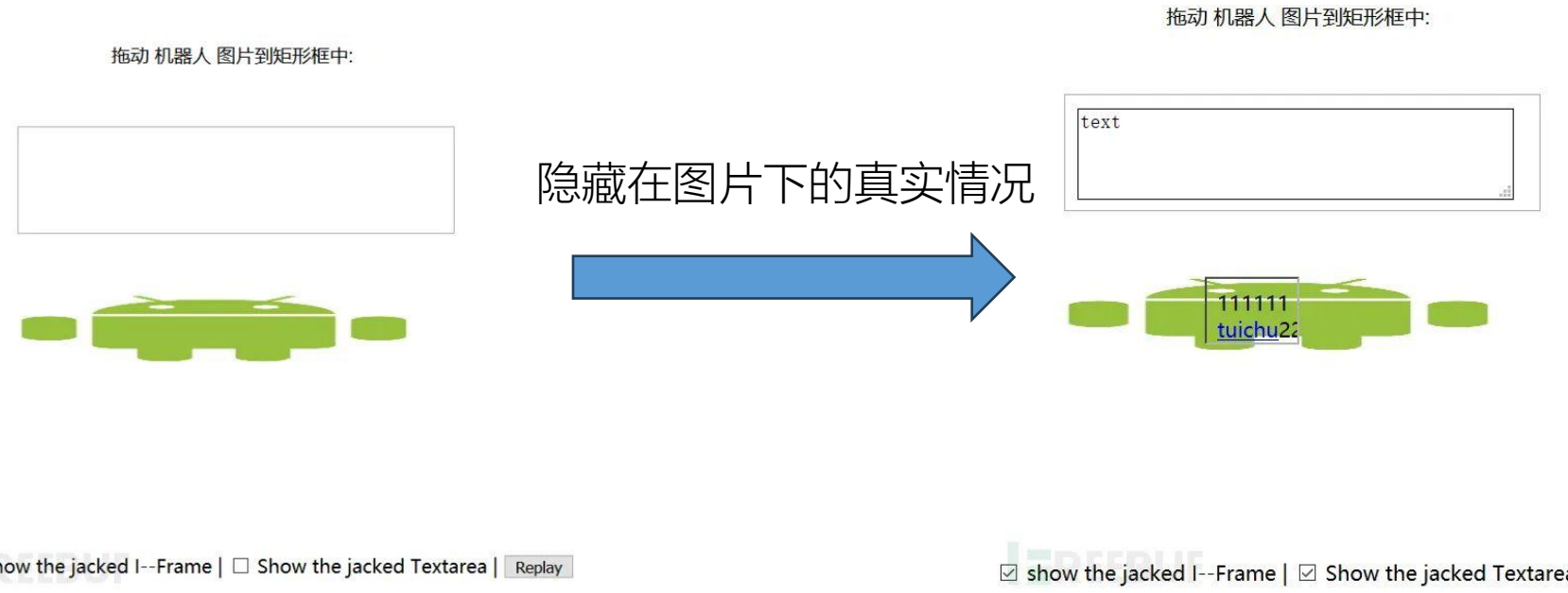
XSS插入的恶意内容

```
259        </div>
260        <iframe src='dashboard.php' id='hack'></iframe>
261
```

点击劫持的类型

- 拖拽劫持与数据窃取

目前很多浏览器都开始支持Drag&Drop的API。对于用户来说，拖拽使他们的操作更加简单。浏览器中的对象可以是链接，文字或者窗口，但是现在拖拽已经被浏览器同源策略所限制，这种攻击已经无效了。



点击劫持的类型

- 拖拽劫持与数据窃取

拖拽劫持的思路是诱使用户从隐藏的不可见iframe中拖拽出攻击者希望得到的数据，然后放到攻击者能控制的另一个页面中，进行窃取数据。

点击劫持的类型

- 新型点击劫持：触屏劫持

一次触屏操作，可能会对应以下几个事件：

- touchstart 手指触摸屏幕时发生
- touchend 手指离开屏幕时发生
- touchmove 手指滑动时发生
- touchcancel 系统可取消touch事件

这种浏览器中的触屏劫持与点击劫持类似

点击劫持的类型

- 新型点击劫持：触屏劫持

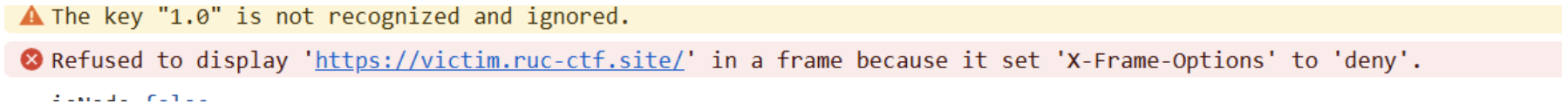
在安卓APP中，恶意APP可以通过钓鱼的方式让你点击某个按钮（对应页面上的特定位置），如果该位置是一个恶意链接，会造成点击劫持

点击劫持的防御

- 可以在 HTTP 响应头中设置 X-Frame-Options 属性，从而控制自己的网站是否可以在 iframe 中显示，例如设置为 DENY 表示不允许任何域加载该资源，SAMEORIGIN 表示仅允许同源请求加载，可以有效防止点击劫持攻击。
- 访问https://victim.ruc-ctf.site/click-jacking/dashboard.php，选择X-Frame-Options选项，登录teacher/teacher。
- 访问钓鱼网页https://hacker.ruc-ctf.site/click-jacking/，可以看到无法加载子页面

```
89          case 1:
90              header("X-Frame-Options: DENY");
91              break;
```

服务端代码



点击劫持的防御

- 设置内容安全策略（CSP）：CSP中有一个frame-ancestors项，表示这个页面在frame tree上的祖先可以来自哪些源，将其设置为self，表示只有当前源可以嵌入，none表示不能被嵌入iframe。
- 访问 https://victim.ruc-ctf.site/click-jacking/index.php ，选择CSP，登录teacher/teacher。
- 访问钓鱼页面https://hacker.ruc-ctf.site/click-jacking/，发现页面无法加载。

服务端防御代码

```
92          case 2:
93              header("Content-Security-Policy: frame-ancestors 'self'");
94              break;
```

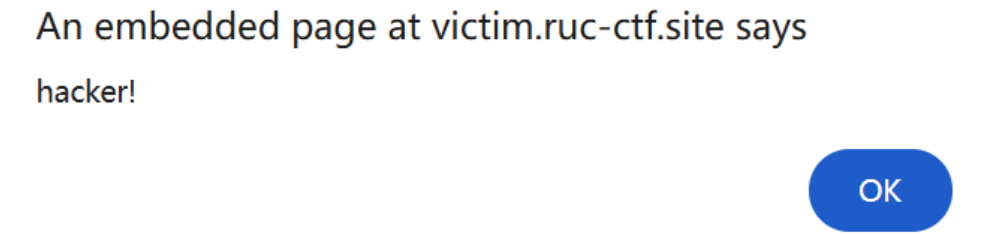
Refused to frame 'https://victim.ruc-ctf.site/' because an ancestor violates the following Content Security Policy directive: "frame-ancestors 'self'".

点击劫持的防御

- 使用Javascript判断当前页面是否被嵌套在其他页面中。
- 访问https://victim.ruc-ctf.site/click-jacking/index.php，选择javascript，登录teacher/teacher。
- 访问钓鱼页面，发现javascript代码检测出页面被嵌入其他页面。

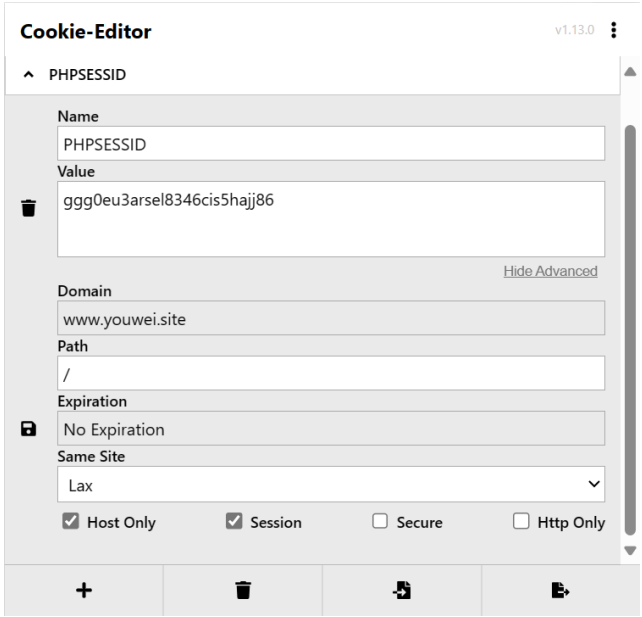
```
95          case 3:
96              echo "<script>
97              if (window.top !== window.self) {
98                  alert('hacker!');
99                  location.href='about:blank_';
100             }
101             </script>
102             ";
103             break;
```

服务端防御代码



点击劫持的防御

- 现在很多站点的操作都需要登录才能完成，网站可以通过禁止第三方站点传输自己的cookie，使得自己的页面被插入在iframe中时，总是未登录状态。
- 打开浏览器的cookie管理插件EditThisCookie(Chrome)、editThisCookie2(FireFox)、Cookie-Editor(Edge)，可以看到cookie的一些属性如图



点击劫持的防御

- 其中same site属性表示cookie是否允许被第三方网站传输
- 注意：并不是说cookie被第三方网站获取了，而是第三方网站对该网站发出的请求是否能附带该网站的cookie
- Same site有三个值：
  - Strict 完全禁止第三方 Cookie，跨站点时，任何情况下都不会发送 Cookie。
  - Lax 允许部分第三方请求携带 Cookie。
  - None 无论是否跨站都会发送 Cookie。
- 如果网站没有指定cookie的same site，浏览器默认行为是Lax
- 特殊地，如果网站试图将自己的cookie的same site值设置为None，则必须同时设置Security属性为true，也就是cookie需要通过https协议传输

点击劫持的防御

same site属性区分				
请求类型	实例	Strict	Lax	None
链接		不发送	发送	发送
预加载	<link rel="prerenderer" href="..."/>	不发送	发送	发送
get表单	<form method="get" action="...">	不发送	发送	发送
post表单	<form method="post" action="...">	不发送	不发送	发送
iframe	<iframe src="..."></iframe>	不发送	不发送	发送
AJAX	$.get("...")	不发送	不发送	发送
image	<img src="...">	不发送	不发送	发送

点击劫持防御实例

- 登录微人大登录 - 中国人民大学 (ruc.edu.cn)，可以看到其cookie的same-site属性都为默认设置，浏览器将其解释为Lax

名称	值	Domain	Path	Expir...	大小	HttpOnly	Secure	SameSite
access_token	22...	.ruc.edu.cn	/	2024...	34	✓		
is_simple	1	v.ruc.edu.cn	/	2024...	10	✓		
session	ba9...	v.ruc.edu.cn	/	会话	72	✓		
tiup_uid	611...	v.ruc.edu.cn	/	会话	32			

- 访问 https://hacker.ruc-ctf.site/click-jacking/iframe-vruc.html，可以看到iframe中的微人大总是未登录状态，即使在这里输入账号密码，也无法登录。

将vruc插入iframe中，查看其登录状态

扩展阅读

- Soheil Khodayari, Giancarlo Pellegrino. JAW: Studying Client-side CSRF with Hybrid Property Graphs and Declarative Traversals. In Proceedings of USENIX SECURITY 2021.
- Jianjun Chen, Jian Jiang, Haixin Duan, Tao Wan, Shuo Chen, Vern Paxson, Min Yang. We Still Don’ t Have Secure Cross-Domain Requests: an Empirical Study of CORS. In Proceedings of USENIX SECURITY 2018.