这个模块主要从7个方面简单的介绍了服务端漏洞.包括:

路径遍历漏洞,访问控制漏洞,身份验证漏洞,服务端请求伪造SSRF,文件上传漏洞,操作系统命令注入和SQL注入

# 1 路径遍历漏洞 :

比如存在一个标签<img src="/loadImage?filename=218.png">

如果没有对文件读取做出过滤,那么在知道文件目录的情况下,也可以尝试:

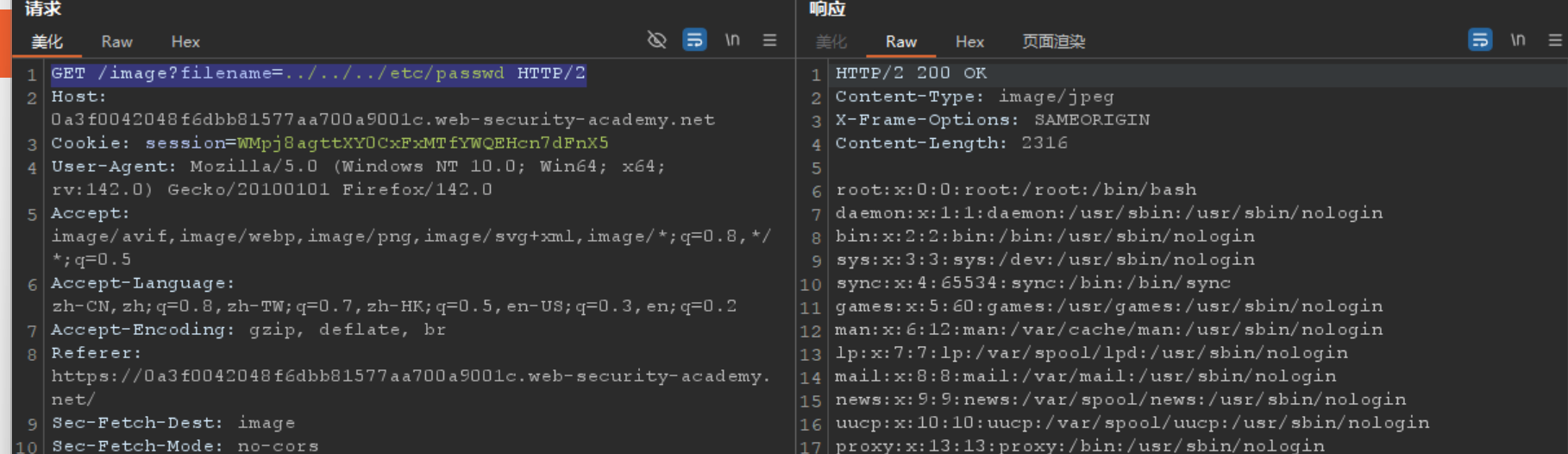
/var/www/images/218.png

那么就可以通过url来读取任意文件,img标签暴露了此漏洞:

<https://insecure-website.com/loadImage?filename=../../../etc/passwd>

例如题目,再访问网站是,图片都是同个GET /image?filename=1.jpg获取的

此处存在路径遍历漏洞:



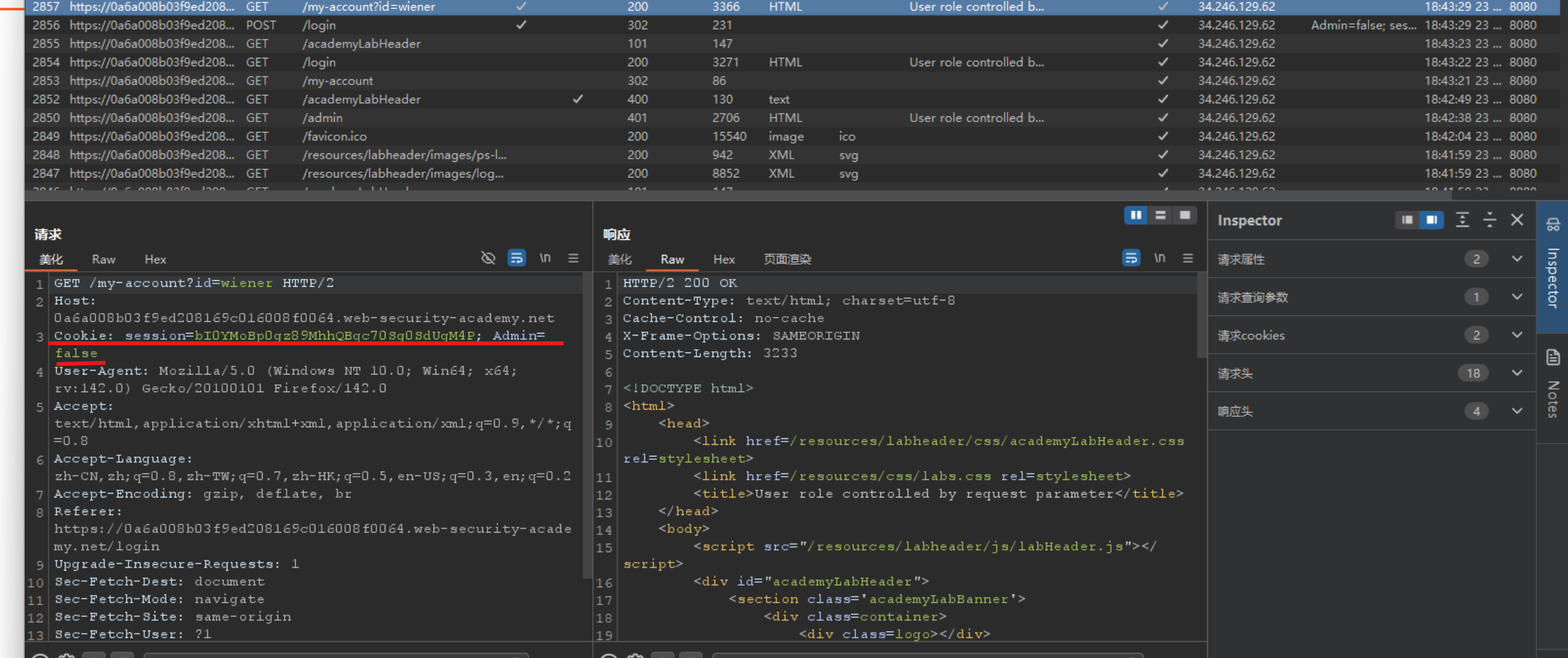
出现这种情况的原因:

未过滤特殊路径符号,未限制路径的 “根目录范围”,依赖客户端校验，忽略服务端校验,路径拼接顺序错误,未处理编码绕过

# 2 访问控制

访问控制漏洞一般指的是垂直权限提升和水平权限获取,最基本的垂直权限提升发生在应用程序没有对敏感信息,功能做出任何保护,任何人都能够访问敏感信息, 1) 比如url/admin直接进入管理页面

1. 有时候,会通过一个不太可能被预测的url来隐藏敏感信息,但是这种敏感信息还是可能会暴露才 js中或者其他地方
2. 有时,可能通过一些参数等来限制管理页面的访问,但是也存在绕过的坑可能,例如:



可以看到,普通用户登录的时候,通过抓包发现,url是可预测的并且cookie中存在敏感信息,可以看到,通过更改cookie中admin=true,

普通用户登录后会出现管理功能:



1. 水平越权:水平越权也常常发生在url可预测的情况下,,但是可能 有些时候页面会使用uuid,guid等防止url可预测,但是,uid仍可能暴露再页面的某些地方,如评论等

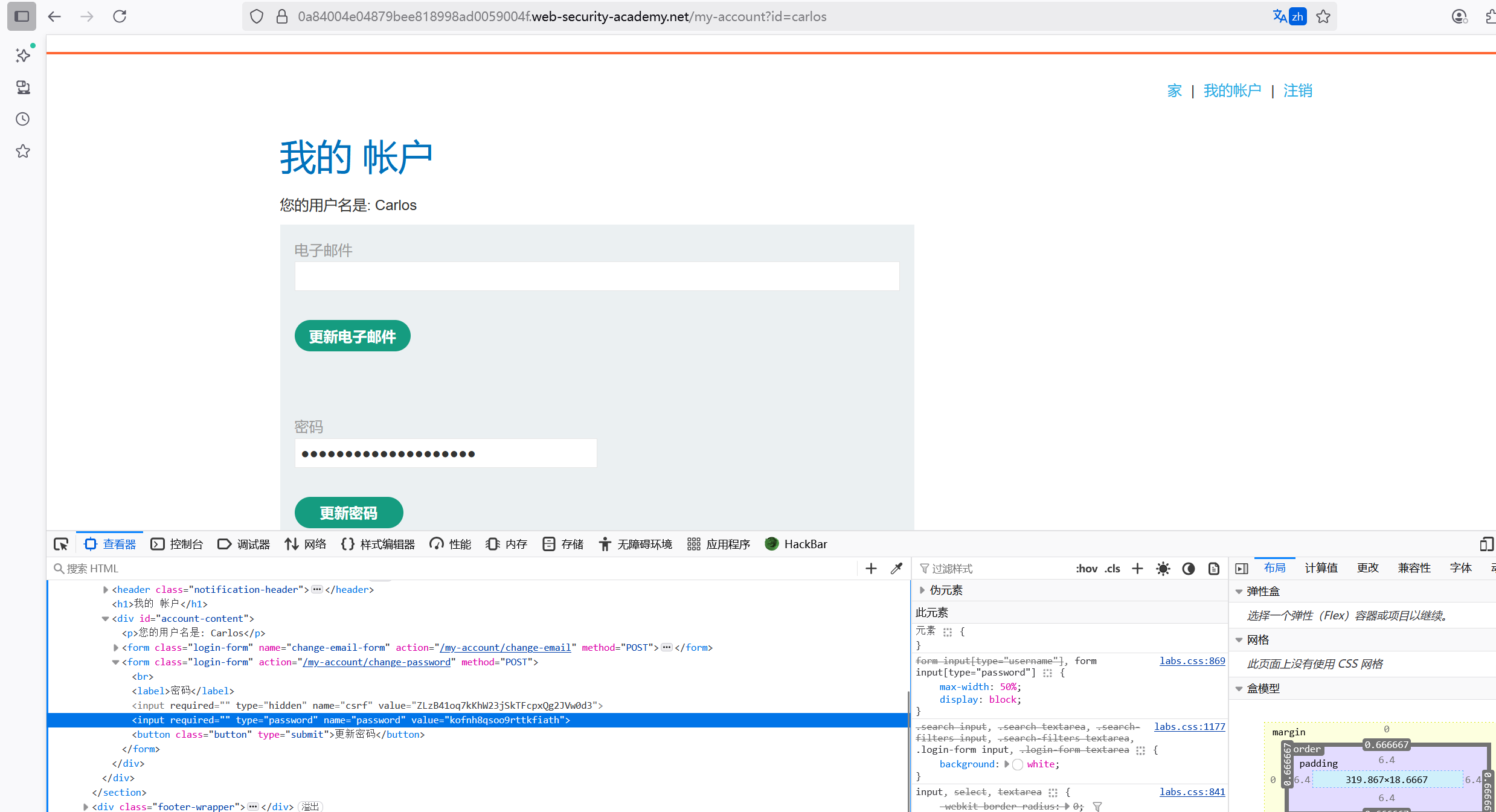


图片中的情况就是 用户个人发布的信息暴露了其uid

1. 通常,横向特权升级攻击可以通过损害更特权的用户而变成垂直特权升级。例如,水平升级可能允许攻击者重置或捕获属于其他用户的密码。

如实验:

首先可以通过url水平越权,得到有用信息:



尝试更改id=administrator

获取管理员密码,登录账号获取权限

# 3 身份验证漏洞

在身份验证的过程中,一个常见的漏洞就是弱口令,因此,暴力破解是一个不错的方法,或者说,默认密码.

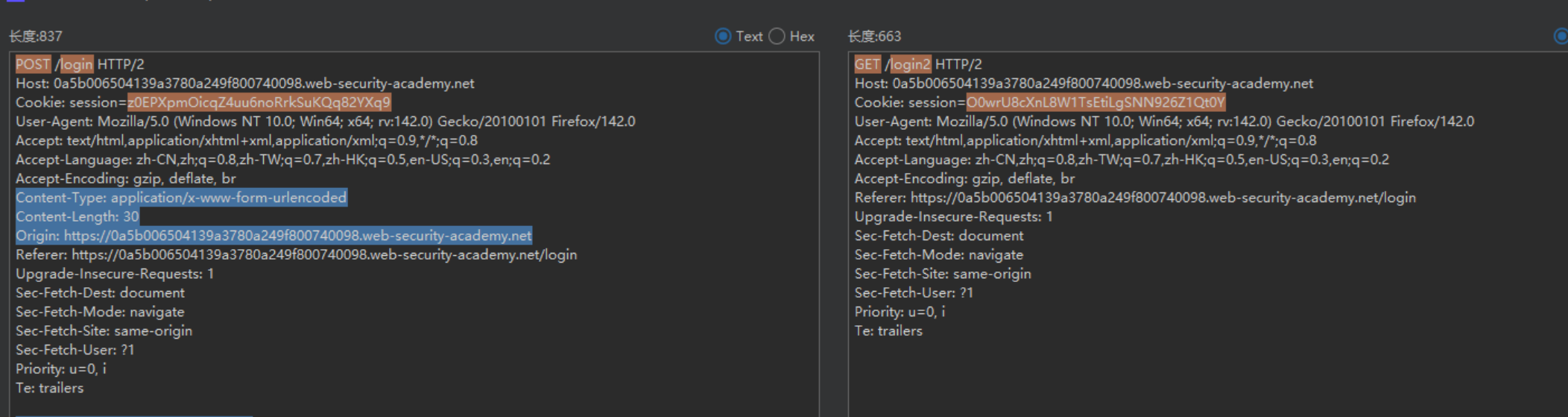
但是有些网站强制要求密码带有特殊符号,数字或者大写等,用户可能会再常用密码中做一些小的调整.,比如:

Admin123!

Wienerpeter1!等,最好能够获取足够的信息,根据信息去破解密码

2 双重身份验证,一些双重身份验证的情况可能要求输入用户名或者密码后 怎增加额外的验证方式,但是配置不当仍有可能出现漏洞

例如,对比登录时候的两个页面:

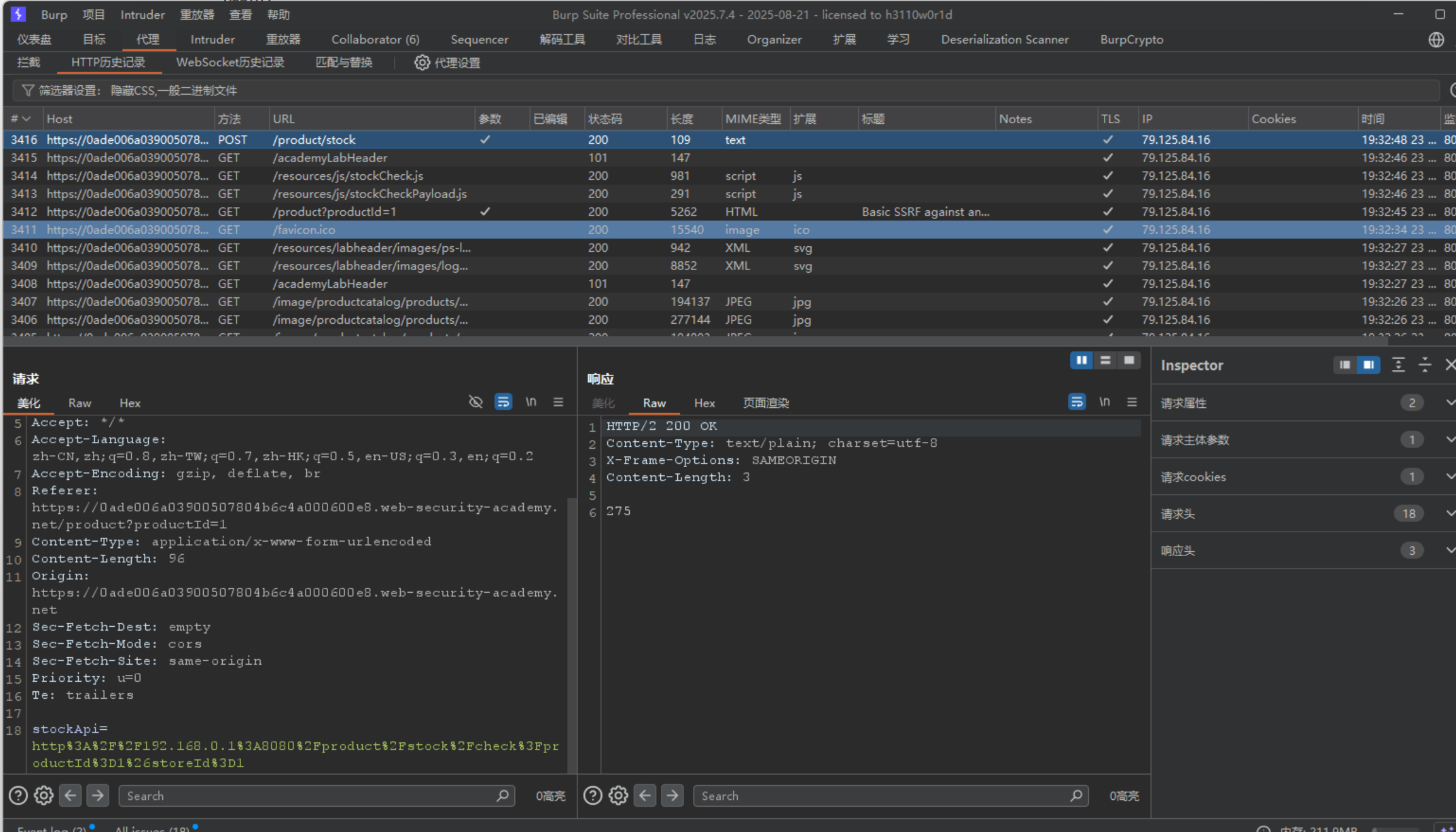


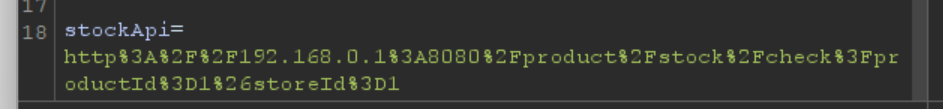
输入正确的用户名和密码后,cookie 发生了改变,其实已经获取了登录凭证,通过更改cookie 或者再另一个标签打开此页面都可以成功的登录 ,或者再需要双重验证的凭证的地方,通过更改url导航到成功登录页面,因为已经过去了有效的cookie不会被拦截

# 4 SSRF 服务端请求伪造

服务器端请求伪造是一个Web安全漏洞,允许攻击者导致服务器端应用程序向意外位置发出请求。在典型的 SSRF 攻击中,攻击者可能导致服务器连接到组织基础架构内的内部服务。在其他情况下,它们可能能够强制服务器连接到任意外部系统。这可能会泄露敏感数据,例如授权凭据。

1. 攻击者可以访问/adminURL,但管理功能通常只对经过身份验证的用户访问。这意味着攻击者不会看到任何感兴趣的东西。但是,如果请求/adminURL来自本地机器,正常的访问控制被绕过。
2. 对于本机机器和外部机器对相同请求的处理方式不同,常常会产生SSRF漏洞





通过抓包可以看到,再库存请求的时候调用了api接口访问内部数据,此处存在ssrf漏洞,通过更改api可以实现对服务器请求的伪造

# 5 文件上传漏洞

文件上上传漏洞一般是对文件上传验证不严格,和一些不同中间件等对文件的解析方式不同导致的,同时网络数据包中的文件类型检查也会被更改导致绕过

<?php echo file\_get\_contents('/path/to/target/file'); ?>

<?php echo system($\_GET['command']); ?>

文件上传漏洞一个重要的信息就是文件的上传后的路径是什么,

可能很对时候网站并不会告诉你文件的路径是怎样的

可以通过上传正确文件 (头像),返回页面 抓包查看 图片是从哪个路径加载的来获取

开发过程中文件路径可能会经过以下设置:

开发侧获取 / 确定最终路径的流程

以 “用户上传头像” 为例，完整路径生成逻辑如下：

服务器接收上传请求，先验证文件合法性（类型、大小、是否恶意文件）；

确定基础根目录：/var/www/upload\_files/（硬编码在后端代码中，不可变）；

生成动态子目录：基于当前日期和用户 ID，如 user\_456/20240823/；

拼接完整路径：/var/www/upload\_files/user\_456/20240823/；

生成唯一文件名（避免覆盖）：如 avatar\_456\_1692789012.png（用户 ID + 时间戳 + 原后缀）；

最终存储路径：/var/www/upload\_files/user\_456/20240823/avatar\_456\_1692789012.png。

一些获取方式:

场景 1：响应体返回路径

场景 2：前端页面显示路径

场景 3：错误信息泄露路径

或者通过”探针”获取

# 6 操作系统命令注入

操作系统命令注入也称为shell注入。它允许攻击者在运行应用程序的服务器上执行操作系统(OS)命令,并且通常完全危及应用程序及其数据。通常,攻击者可以利用操作系统命令注入漏洞来破坏托管基础设施的其他部分,并利用信任关系将攻击转向组织内的其他系统。

系统注入漏洞需要 探索后端 的运行逻辑

需要发现哪里存在使用系统命令的地方,

示例中,购物应用程序允许用户查看特定商店中是否有库存的物品。这些信息通过URL访问:

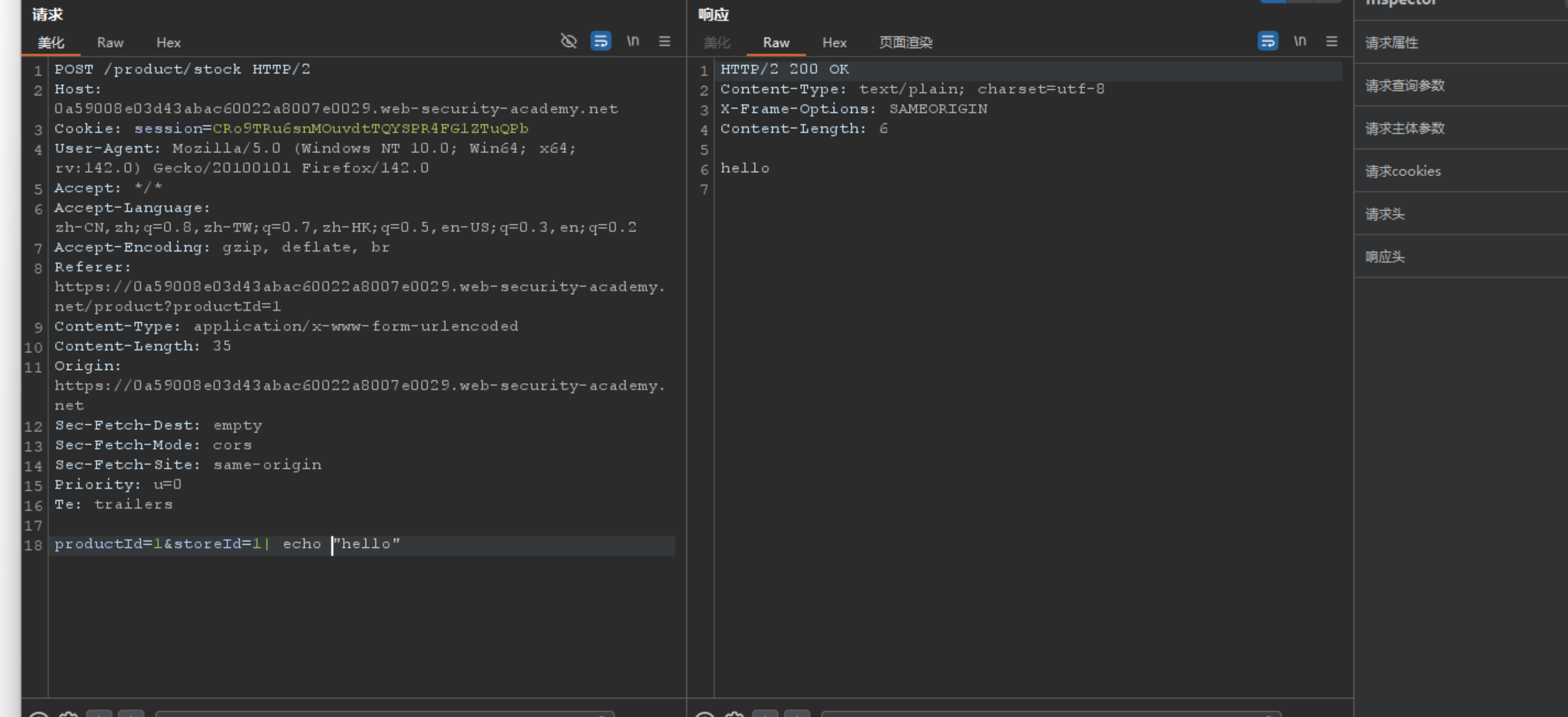
https://insecure-website.com/stockStatus?productID=381&storeID=29

要提供库存信息,应用程序必须查询各种遗留系统。由于历史原因,该功能通过调用带有产品和存储ID作为参数的shell命令来实现:

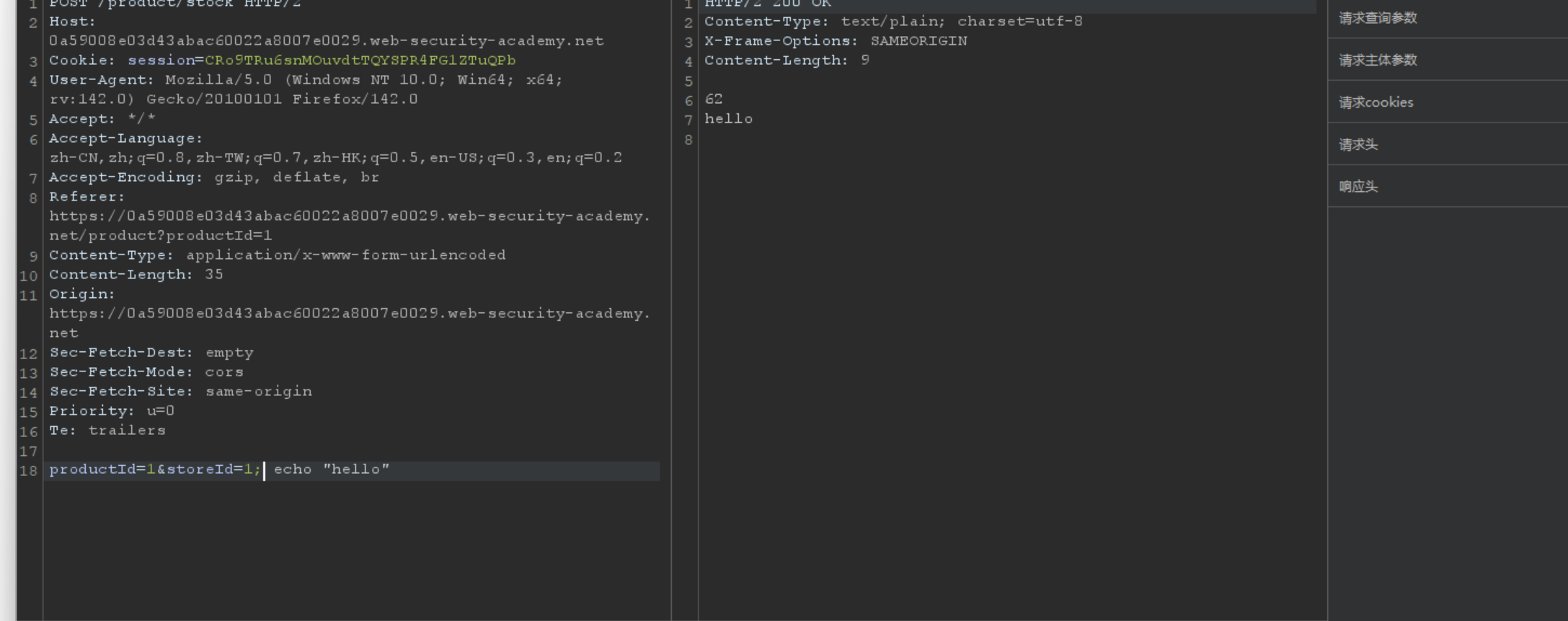
stockreport.pl 381 29

那么这里就存在系统注入漏洞

比如案例中,



通过测试存在命令注入漏洞



通过 “|” “;” 等拼接其他命令并执行

这是存在回显的情况,可能一切不存在回显的地方也能执行此漏洞

可以编写恶意文件,执行非法操作等

命令注入的根源：

服务器将用户可控的输入（如文件名、路径）直接拼接进系统命令，未进行严格的过滤、转义或白名单校验。例如，未过滤;、&、|、&&、||、$()、`等 shell 特殊字符。

需要严格过滤和转义\

# 7 sql注入

这个比较熟了,不想写了