

RPC过某直聘zp_token

0背景

因为最近在求职，想要更加的明确公司的招聘需求，所以想着写一个爬虫爬取招聘信息中的用人要求做个整合分析，在分析接口数据的时候，在/wapi/zpgeek/search/joblist.json这个接口中找到了我们想要的基础招聘数据，所以基于这个接口进行了一些分析。

1逆向思路

在对某个接口进行逆向分析时，首先要看接口中哪些字段可能是动态的加密字段，哪些字段是可以固定下来重复使用的。并且更改请求参数进行测试，在参数变更的情况原先的加密参数是否还能使用(是否对请求参数进行签名),找到加密字段后再对该字段进行分析。

2加密分析

复制完整的请求到python中进行请求，用控制变量的方式，逐个将可疑的变量删除，直到返回异常信息，最终发现加密的字段是Cookie中的zp_token这个字段，一段时间后该token就会自动失效需要重新生成。


3逆向分析

知道加密的字段是zp_token后，我们需要找到它的生成逻辑、生成位置，即知道它是怎么来的。


3.1Hook Cookie

通过注入代码对指定的Cookie进行Hook，当生成或更新该Cookie时自动断点，快速定位到生成的代码，Hook的原理就是通过代理document.cookie方法，当js使用该方法设置cookie时先执行我们的逻辑加个debugger在执行其原本的添加Cookie的逻辑。这边使用油猴注入Hook代码，然后使用网上的Hook插件进行Hook。

3.2断点调试

此时再刷新页面就会停在我们的debugger中，通过右侧的Call Back查看调用堆栈即可找到生成逻辑的位置。

最终跟到他的js，发现这边将a变量的值插入了cookie，查看a的值果然是我们要找的token值,往上翻a是咋来的。

image-20241022150012638

找到一条关键代码,e的值是zp_token,那么这条代码就是拼接Cookie的逻辑了，后面的encodeURIComponent(t)就是我们想要的加密内容，可以发现t是通过方法传入的参数，我们要继续往上跟找到传参的位置才能知道t是咋来的。在这个地方下一个断点进行分析。

这边有个坑，多次刷新会发现后端返回的js文件不同，这导致我们每次下的断点，之后刷新并不能正确的停在我们想要的地方，所以我们需要固定相关的js，所有的静态资源都是从首个静态文件调用的，所以需要固定第一个放回的静态页面，固定后他里面调用的js就是固定的了。这样就能保证服务器每次返回的都是我们调试的js了。注：固定文件使用的是浏览器的overwrite功能。

```
var a = e + "=" + encodeURIComponent(t);
```

刷新界面后继续往上跟，可以发现t是这边的r变量传过去的，继续在这边找r变量是怎么来的。

 image-20241022151010281

往上翻找到了r的生成位置，这边就是我们的关键代码，如果是要扣算法的话害得继续跟z的逻辑，但是因为我们使用rpc的方式，只需要找到他的生成位置即可。我们固定这条js，在其下方插桩注入我们的代码，将这个生成r的方法注入到一个全局的方法里面去，当我们刷新页面，代码运行到这边时就会将这个逻辑执行，我们就可以在任意的地方调用这个代码生成加密内容了。


```
// 传入需要的参数，返回加密内容
window.boss = function name(seed, ts) {
    return (new e).z(seed, parseInt(ts) + 60 * (480 + (new
Date).getTimezoneOffset()) * 1e3)
}
```

 image-20241022151306037

但是这边还有两个参数是我们不知道的一个是t，一个是n，对应我们的参数就是seed和ts。他们是参数传递进来的，所以我们还要往上跟找到他们是啥。最终在上一个方法中找到了他们其实是从Cookie中分别取了zp_sseed和zp_sts这两个Cookie值，这也是为啥我们的参数要命名seed和ts的原因。

 image-20241022152006837

知道了seed和ts是从Cookie中取的，但是我们要知道他不会一直就在Cookie里，肯定也是在哪边设置的，最后在反复的查找中发现。当我们使用该接口请求成功后，响应头中会对其值进行设置，当我们请求失败他会直接在响应体中返回值。到这我们的逆向分析就结束了。

 image-20241022152516000

3RPC调用

3.1调用测试

上面我们成功找到了加密字段zp_stoken，并插桩让我们可以在任意位置调用其生成逻辑，这边我们测试一下调用。在浏览器控制台调用我们注入的方法并传入对应的参数，发现成功调用生成逻辑，并返回加密内容。

 image-20241022153119395

通过上面的分析我们可以知道，最终加密内容还会通过encodeURIComponent进行编码，我们将编码后的值填充到Cookie中发起请求进行测试。成功返回我们想要的的数据。

 image-20241022153858493

3.2Sekiro RPC框架

这边的sekiro可不是打铁的只狼，是一个提供远程调用的rpc框架。只需要启动他的服务端，然后在需要远程调用的位置使用提供的API注册服务，我们就可以在任意位置调用该服务。

我们使用油猴注入代码到网页中，将注册服务的代码注入到当前的网页。这边有几个重要的点，一个是group，一个是传入的方法名，待会我们远程调用这个服务的时候会用到，可以看到在下面的逻辑中调用了我们之前插桩的函数，调用那个函数生成加密内容进行返回，即可通过远程调用的方式拿到加密的内容。

```
function SekiroClient(e){
    //... 省略内容，这边有一长串的代码，在sekiro文档中有
    var client = new SekiroClient("ws://127.0.0.1:5612/business/register?
group=boss&clientId=" + Math.random());
    client.registerAction("getZpStoken", function (request, resolve, reject) {
```

```

    try{
      var seed = request['seed'];
      var ts = request['ts'];
      console.log("rpc调用,seed:"+seed+",ts:"+ts);
      var result = encodeURIComponent(window.boss(seed, ts));
      resolve(result);
    }catch(e){
      reject("error ->> "+e)
    }
  });

```

上面的代码向sekiro服务器中注册了一个方法名为 getZpStoken的服务，这时我们只需要向sekiro服务器发送请求调用该服务即可。参数中的group和action要与上面注册的时候一致。再传入需要的参数即可完成远程调用拿到浏览器生成的加密内容。

```

params = {
  'group': 'boss',
  'action': "getZpStoken",
  'seed': seed,
  'ts': ts
}
rsp = session.get('http://127.0.0.1:5612/business/invoke', params=params)
data = rsp.json()

```

 image-20241022155258776

4调用逻辑分析

在调用接口请求数据时，我们首先要拿到seed和ts的值，但是因为这两项是在请求之后才会变更，我们可以先生成一个正确的token值进行调用，调用成功后会在响应头返回变更后的值，再拿到变更的值调用远程服务去获取加密的token值进行调用，反复循环，即可爬取页面上的所有内容。

 image-20241022161253779

5结语

这之后就可以将清洗的数据存储到ES或者数据库中进行后续的分析了。但是这边其实还有个坑，他会禁IP，不知道他的检测策略，估计是调用频繁之后就会封禁IP，之后该IP再发起请求就会异常了。当然这边也可以使用代理IP池的方式轮换IP防止检测，毕竟道高一尺，魔高一丈。