RESTful架构风格下的4大常见安全问题

马伟 分类： 新兴技术

伴随着RESTful架构风格的大量应用微服务架构的流行，一些本来难以察觉到的安全问题也逐渐开始显现出来。在我经历过的各种采用RESTful微服务架构风格的应用中，某些安全问题几乎在每个应用中都会出现。然而它们并非是什么高深的技术难题，只不过是借着微服务的流行而显得越发突出，这些都可以通过一些安全实践来避免。本文将一些典型的问题列举出来，希望能引起开发团队的注意，帮助他们绕过这些安全问题的“坑”。

1. 遗漏了对资源从属关系的检查

一个典型的RESTful的URL会用资源名加上资源的ID编号来标识其唯一性，就像这样: `/users/<USER ID>` ， 例如： `/users/100`

一般而言用户只能查看自己的用户信息，而不允许查看其它用户的信息。在这种情况下，攻击者很可能会尝试把这个URL里面的USER ID从100修改为其他数值，以期望应用返回指定用户的信息。不过由于这个安全风险太显而易见，绝大多数应用都会对当前请求者的身份进行校验，看其是否是编号为100的用户，校验成功才返回URL中指定的用户信息，否则会拒绝当前请求。

对于URL中只出现一个资源的情况，绝大多数应用都已经做了安全防御，然而重灾区出现在URL中包含多个资源的时候。

以用户查看订单的RESTful URL为例：`/users/100/orders/280010`，应用只检查了当前请求发起者是否是编号为100的用户，以及编号为280010的订单是否存在，有很大的概率没有检查URL中的订单和用户之间的从属关系。其结果是，攻击者可以通过修改URL中的订单编号，从而遍历系统中的所有订单信息，甚至对不属于他/她的订单发起操作，例如取消订单。

上面的例子中只有两个资源，如果URL中资源数量继续增加，这种从属关系校验缺失的情况只会更加普遍。

解决这一问题的方法极其简单，只要发现URL里面出现了两个或者两个以上的资源，就像下面这样：

```
/ResourceA/<ResourceA Id>/ResourceB/<ResourceB Id>/ResourceC/<ResourceC Id>
```

在对资源进行操作之前，就得先检查这些资源之间的从属关系，以确保当前请求具有相关的访问、操作权限。

2. HTTP响应中缺失必要的 Security Headers

HTTP中有一些和安全相关的Header，通过对它们的合理使用，可以使得应用在具备更高的安全性的同时，并不会显著增大开发者的工作负担，有着“低成本高收益”的效果。不过绝大多数情况下，这些Header是默认关闭的，因此很多应用中也就缺失了这些Security Headers。一些典型的Security Headers如下：

X-Frame-Options

为了防止应用遭受点击劫持攻击，可以使用 `X-Frame-Options: DENY` 明确告知浏览器，不要把当前HTTP响应中的内容在HTML Frame中显示出来。

X-Content-Type-Options

在浏览器收到HTTP响应内容时，它会尝试按照自己的规则去推断响应内容的类型，并根据推断结果执行后续操作，而这可能造成安全问题。例如，一个包含恶意JavaScript代码的HTTP响应内容，虽然其 `Content-Type` 为 `image/png` ，但是浏览器推断出这是一段脚本并且会执行它。

`X-Content-Type-Options` 就是专门用来解决这个问题的Header。通过将其设置为 `X-Content-Type-Options: nosniff` ，浏览器将不再自作主张的推断HTTP响应内容的类型，而是严格按照响应中 `Content-Type` 所指定的类型来解析响应内容。

X-XSS-Protection

避免应用出现跨站脚本漏洞（Cross-Site Scripting，简称XSS）的最佳办法是对输出数据进行正确的编码，不过除此之外，现如今的浏览器也自带了防御XSS的能力。

要开启浏览器的防XSS功能，只需要在HTTP响应中加上这个Header：`X-XSS-Protection: 1; mode=block`。其中，数字 1 代表开启浏览器的XSS防御功能，`mode=block` 是告诉浏览器，如果发现有XSS攻击，则直接屏蔽掉当前即将渲染的内容。

Strict-Transport-Security

使用TLS可以保护数据在传输过程中的安全，而在HTTP响应中添加上 `Strict-Transport-Security` 这个Header，可以告知浏览器直接发起HTTPS请求，而不再像往常那样，先发送明文的HTTP请求，得到服务器跳转指令后再发送后续的HTTPS请求。并且，一旦浏览器接收到这个Header，那么当它发现数据传输通道不安全的时候，它会直接拒绝进行任何的数据传输，不再允许用户继续通过不安全的传输通道传输数据，以避免信息泄露。

3. 不经意间泄露的业务信息

会说话的ID

资源ID是RESTful URL中很重要的一个组成部分，大多数情况下这类资源ID都是用数字来表示的。这在不经意间泄露了业务信息，而这些信息可能正是竞争对手希望得到的数据。

以查看用户信息的RESTful URL为例：`/users/100`。由于用户ID是一个按序递增的数字，因此攻击者既可以通过ID知道目前应用中的用户规模，也可以分别在月初和月末的时候注册一个用户，并对比两个用户的ID即可知道当前这个月有多少新增用户。同理，如果订单号也是按序自增的数字，攻击者可以了解到一定时间范围内的订单量。

这类ID并不会给应用造成任何技术上的威胁，只是通过ID泄露出来的信息对于你的业务而言可能非常敏感。解决办法是不使用按序递增的数字作为ID，而是使用具有随机性、唯一性、不可预测性的值作为ID，最常见的做法就是使用UUID。

返回多余的数据

前后端分离的情况下，两者之间通常以JSON作为数据传输的主体。有时候可能是为了方便前端代码处理，也可能是疏忽大意，总之后端API返回的JSON数据中包含了远远超出前端代码需要的数据，因此造成数据泄露。

例如，前端代码本意是请求订单信息，但是后端API返回的订单JSON数据中还包含了很多“有意思”的数据。

```
{
    "id": 280010,
    "orderItems": [...],
    "user": {
        "id": 100,
        "password": "91B4E3E45B2465A4823BB5C03FF81B65"
    },
    ...
}
```

上面这个例子里，订单数据中包含了用户信息，最为关键的是连用户的密码字段也被包含在内。

解决办法显而易见，在给前端返回数据之前，将这些敏感的、前端并不需要的数据过滤掉。技术上实现起来易如反掌，但是真正难的地方在于让整个应用都严格的按照这样的方式来处理JSON数据，确保没有任何遗漏之处。

4. API缺乏速率限制的保护

先看一个例子。用户注册时发送短信验证码的API，由于没有做速率限制，使得攻击者可以用一段脚本不断的请求服务器发送短信验证码，导致在短时间内耗尽短信发送配额，或者造成短信网关拥挤等等后果。

受伤的不仅仅是发送短信的API，其他一些比较敏感的API如果缺乏请求速率限制的保护，同样也会遭遇安全问题。例如用户登录的API缺乏速率限制的话，攻击者可以利用其进行用户名密码暴力破解，再例如某些大量消耗服务器资源的API如果缺乏速率限制，攻击者可以利用其发起拒绝式攻击。

解决这类安全问题的原则就是对API请求的速率进行适当的限制。具体的做法有很多，最典型的例子就是使用图片验证码，其他的做法还有利用Redis的Expire特性对请求速率进行统计判断，甚至借助运维的力量（例如网络防火墙）来共同进行防御等等。

总结

开发出一个具备足够安全性的应用不是件容易的事情，本文中提到的只是RESTful架构风格下，众多安全问题中比较典型的一部分而已。之所以会有这些问题，其本质原因在于应用开发过程中，开发团队的注意力集中在业务功能的实现上，应用安全性相关的需求没有得到足够的明确和重视。

如果你不想被这些安全问题所困扰，建议通过在应用开发过程中引入威胁建模、在用户故事卡中设立安全验收标准、进行安全代码审查等一系列安全实践，尽可能从源头上规避这些问题。

更多精彩洞见，请关注微信公众号：思特沃克