

## 27 | 数据源头：任何客户端的东西都不可信任

2020-05-19 朱晔

Java业务开发常见错误100例

[进入课程 >](#)



**讲述：王少泽**

时长 16:31 大小 15.13M



你好，我是朱晔。

从今天开始，我要和你讨论几个有关安全的话题。首先声明，我不是安全专家，但我发现有这么一个问题，那就是许多做业务开发的同学往往一点点安全意识都没有。如果有些公司没有安全部门或专家的话，安全问题就会非常严重。

如果只是用一些所谓的渗透服务浅层次地做一下扫描和渗透，而不在代码和逻辑层面做进一步分析的话，能够发现的安全问题非常有限。要做好安全，还是要靠一线程序员和产品点



点点点滴滴的意识。

所以接下来的几篇文章，我会从业务开发的角度，和你说说我们应该最应该具备的安全意识。

对于 HTTP 请求，我们要在脑子里有一个根深蒂固的概念，那就是**任何客户端传过来的数据都是不能直接信任的**。客户端传给服务端的数据只是信息收集，数据需要经过有效性验证、权限验证等后才能使用，并且这些数据只能认为是用户操作的意图，不能直接代表数据当前的状态。

举一个简单的例子，我们打游戏的时候，客户端发给服务端的只是用户的操作，比如移动了多少位置，由服务端根据用户当前的状态来设置新的位置再返回给客户端。为了防止作弊，不可能由客户端直接告诉服务端用户当前的位置。

因此，客户端发给服务端的指令，代表的只是操作指令，并不能直接决定用户的状态，对于状态改变的计算在服务端。而网络不好时，我们往往会遇到走了 10 步又被服务端拉回来的现象，就是因为有指令丢失，客户端使用服务端计算的实际位置修正了客户端玩家的位置。


今天，我通过四个案例来和你说说，为什么“任何客户端的东西都不可信任”。

## 客户端的计算不可信

我们先看一个电商下单操作的案例。

在这个场景下，可能会暴露这么一个 /order 的 POST 接口给客户端，让客户端直接把组装后的订单信息 Order 传给服务端：

```
1 @PostMapping("/order")
2 public void wrong(@RequestBody Order order) {
3     this.createOrder(order);
4 }
```

 复制代码

订单信息 Order 可能包括商品 ID、商品价格、数量、商品总价：

```
1 @Data
2 public class Order {
```

 复制代码

```
3     private long itemId; //商品ID
4     private BigDecimal itemPrice; //商品价格
5     private int quantity; //商品数量
6     private BigDecimal itemTotalPrice; //商品总价
7 }
```

虽然用户下单时客户端肯定有商品的价格等信息，也会计算出订单的总价给用户确认，但是这些信息只能用于呈现和核对。即使客户端传给服务端的 POJO 中包含了这些信息，服务端也一定要重新从数据库来初始化商品的价格，重新计算最终的订单价格。**如果不这么做的话，很可能会被黑客利用，商品总价被恶意修改为比较低的价格。**


因此，我们真正直接使用的、可信赖的只是客户端传过来的商品 ID 和数量，服务端会根据这些信息重新计算最终的总价。如果服务端计算出来的商品价格和客户端传过来的价格不匹配的话，可以给客户端友好提示，让用户重新下单。修改后的代码如下：

 复制代码

```
1 @PostMapping("/orderRight")
2 public void right(@RequestBody Order order) {
3     //根据ID重新查询商品
4     Item item = Db.getItem(order.getItemId());
5     //客户端传入的和服务端查询到的商品单价不匹配的时候，给予友好提示
6     if (!order.getItemPrice().equals(item.getItemPrice())) {
7         throw new RuntimeException("您选购的商品价格有变化，请重新下单");
8     }
9     //重新设置商品单价
10    order.setItemPrice(item.getItemPrice());
11    //重新计算商品总价
12    BigDecimal totalPrice = item.getItemPrice().multiply(BigDecimal.valueOf(or
13    //客户端传入的和服务端查询到的商品总价不匹配的时候，给予友好提示
14    if (order.getItemTotalPrice().compareTo(totalPrice) != 0) {
15        throw new RuntimeException("您选购的商品总价有变化，请重新下单");
16    }
17    //重新设置商品总价
18    order.setItemTotalPrice(totalPrice);
19    createOrder(order);
20 }
```

还有一种可行的做法是，让客户端仅传入需要的数据给服务端，像这样重新定义一个 POJO `CreateOrderRequest` 作为接口入参，比直接使用领域模型 `Order` 更合理。在设计接口时，我们会思考哪些数据需要客户端提供，而不是把一个大而全的对象作为参数提供给服务端，以避免因为忘记在服务端重置客户端数据而导致的安全问题。

下单成功后，服务端处理完成后会返回诸如商品单价、总价等信息给客户端。此时，客户端可以进行一次判断，如果和之前客户端的数据不一致的话，给予用户提示，用户确认没问题后再进入支付阶段：

 复制代码

```
1 @Data
2 public class CreateOrderRequest {
3     private long itemId; //商品ID
4     private int quantity; //商品数量
5 }
6
7 @PostMapping("orderRight2")
8 public Order right2(@RequestBody CreateOrderRequest createOrderRequest) {
9     //商品ID和商品数量是可信的没问题，其他数据需要由服务端计算
10    Item item = Db.getItem(createOrderRequest.getItemId());
11    Order order = new Order();
12    order.setItemPrice(item.getItemPrice());
13    order.setItemTotalPrice(item.getItemPrice().multiply(BigDecimal.valueOf(ord
14    createOrder(order);
15    return order;
16 }
```

通过这个案例我们可以看到，在处理客户端提交过来的数据时，服务端需要明确区分，哪些数据是需要客户端提供的，哪些数据是客户端从服务端获取后在客户端计算的。其中，前者可以信任；而后者不可信任，服务端需要重新计算，如果客户端和服务端计算结果不一致的话，可以给予友好提示。

## 客户端提交的参数需要校验

对于客户端的数据，我们还容易忽略的一点是，**误以为客户端的数据来源是服务端，客户端就不可能提交异常数据**。我们看一个案例。

有一个用户注册页面要让用户选择所在国家，我们会把服务端支持的国家列表返回给页面，供用户选择。如下代码所示，我们的注册只支持中国、美国和英国三个国家，并不对其他国家开放，因此从数据库中筛选了 `id<4` 的国家返回给页面进行填充：

 复制代码

```
1 @Slf4j
2 @RequestMapping("trustclientdata")
3 @Controller
4 public class TrustClientDataController {
```

```

5    //所有支持的国家
6    private HashMap<Integer, Country> allCountries = new HashMap<>();
7
8    public TrustClientDataController() {
9        allCountries.put(1, new Country(1, "China"));
10       allCountries.put(2, new Country(2, "US"));
11       allCountries.put(3, new Country(3, "UK"));
12       allCountries.put(4, new Country(4, "Japan"));
13   }
14
15   @GetMapping("/")
16   public String index(ModelMap modelMap) {
17       List<Country> countries = new ArrayList<>();
18       //从数据库查出ID<4的三个国家作为白名单在页面显示
19       countries.addAll(allCountries.values().stream().filter(country -> coun
20       modelMap.addAttribute("countries", countries);
21       return "index";
22   }
23 }

```

我们通过服务端返回的数据来渲染模板：

 复制代码

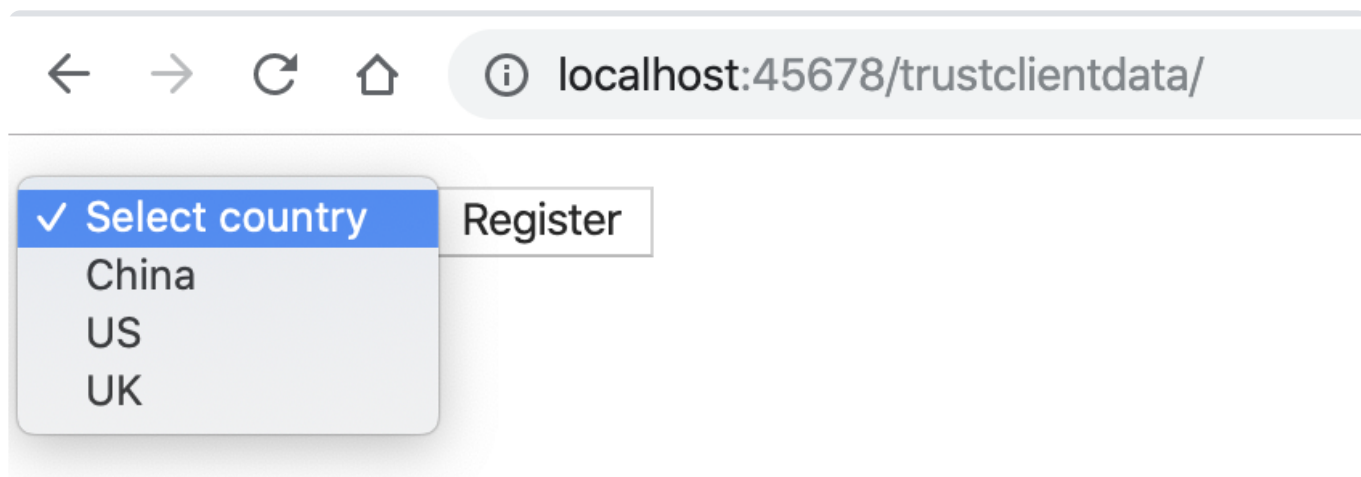
```

1    ...
2    <form id="myForm" method="post" th:action="@{/trustclientdata/wrong}">
3
4
5        <select id="countryId" name="countryId">
6            <option value="0">Select country</option>
7            <option th:each="country : ${countries}" th:text="${country.name}" th:
8        </select>
9
10
11        <button th:text="Register" type="submit"/>
12    </form>
13    ...

```

在页面上，的确也只有这三个国家的可选项：






但我们要知道的是，页面是给普通用户使用的，而黑客不会在乎页面显示什么，完全有可能尝试给服务端返回页面上没显示的其他国家 ID。如果像这样直接信任客户端传来的国家 ID 的话，很可能会把用户注册功能开放给其他国家的人：

 复制代码

```
1 @PostMapping("/wrong")
2 @ResponseBody
3 public String wrong(@RequestParam("countryId") int countryId) {
4     return allCountries.get(countryId).getName();
5 }
```

即使我们知道参数的范围来自下拉框，而下拉框的内容也来自服务端，也需要对参数进行校验。因为接口不一定要通过浏览器请求，只要知道接口定义完全可以通过其他工具提交：

 复制代码


```
1 curl http://localhost:45678/trustclientdata/wrong?countryId=4 -X POST
```

修改方式是，在使用客户端传过来的参数之前，对参数进行有效性校验：

 复制代码

```
1 @PostMapping("/right")
2 @ResponseBody
3 public String right(@RequestParam("countryId") int countryId) {
4     if (countryId < 1 || countryId > 3)
5         throw new RuntimeException("非法参数");
6     return allCountries.get(countryId).getName();
7 }
```

或者是，使用 Spring Validation 采用注解的方式进行参数校验，更优雅：

 复制代码

```
1 @Validated
2 public class TrustClientParameterController {
3     @PostMapping("/better")
4     @ResponseBody
5     public String better(
6         @RequestParam("countryId")
7         @Min(value = 1, message = "非法参数")
8         @Max(value = 3, message = "非法参数") int countryId) {
9         return allCountries.get(countryId).getName();
10    }
11 }
```


客户端提交的参数需要校验的问题，可以引申出一个更容易忽略的点是，我们可能会把一些服务端的数据暂存在网页的隐藏域中，这样下次页面提交的时候可以把相关数据再传给服务端。虽然用户通过网页界面的操作无法修改这些数据，但这些数据对于 HTTP 请求来说就是普通数据，完全可以随时修改为任意值。所以，服务端在使用这些数据的时候，也同样要特别小心。

## 不能信任请求头里的任何内容

刚才我们介绍了，不能直接信任客户端的传参，也就是通过 GET 或 POST 方法传过来的数据，此外请求头的内容也不能信任。

一个比较常见的需求是，为了防刷，我们需要判断用户的唯一性。比如，针对未注册的新用户发送一些小奖品，我们不希望相同用户多次获得奖品。考虑到未注册的用户因为没有登录过所以没有用户标识，我们可能会想到根据请求的 IP 地址，来判断用户是否已经领过奖品。

比如，下面的这段测试代码。我们通过一个 HashSet 模拟已发放过奖品的 IP 名单，每次领取奖品后把 IP 地址加入这个名单中。IP 地址的获取方式是：优先通过 X-Forwarded-For 请求头来获取，如果没有的话再通过 HttpServletRequest 的 getRemoteAddr 方法来获取。

 复制代码

```
1 @Slf4j
```

```


2 @RequestMapping("trustclientip")
3 @RestController
4 public class TrustClientIpController {
5
6     HashSet<String> activityLimit = new HashSet<>();
7
8     @GetMapping("test")
9     public String test(HttpServletRequest request) {
10         String ip = getClientIp(request);
11         if (activityLimit.contains(ip)) {
12             return "您已经领取过奖品";
13         } else {
14             activityLimit.add(ip);
15             return "奖品领取成功";
16         }
17     }
18
19     private String getClientIp(HttpServletRequest request) {
20         String xff = request.getHeader("X-Forwarded-For");
21         if (xff == null) {
22             return request.getRemoteAddr();
23         } else {
24             return xff.contains(",") ? xff.split(",")[0] : xff;
25         }
26     }
27 }

```

之所以这么做是因为，通常我们的应用之前都部署了反向代理或负载均衡器，remoteAddr 获得的只能是代理的 IP 地址，而不是访问用户实际的 IP。这不符合我们的需求，因为反向代理在转发请求时，通常会把用户真实 IP 放入 X-Forwarded-For 这个请求头中。

**这种过于依赖 X-Forwarded-For 请求头来判断用户唯一性的实现方式，是有问题的：**

完全可以通过 cURL 类似的工具来模拟请求，随意篡改头的内容：

 复制代码

```
1 curl http://localhost:45678/trustclientip/test -H "X-Forwarded-For:183.84.18.7"
```


网吧、学校等机构的出口 IP 往往是同一个，在这个场景下，可能只有最先打开这个页面的用户才能领取到奖品，而其他用户会被阻拦。



因此，IP 地址或者说请求头里的任何信息，包括 Cookie 中的信息、Referer，只能用作参考，不能用作重要逻辑判断的依据。而对于类似这个案例唯一性的判断需求，更好的做法是，让用户进行登录或三方授权登录（比如微信），拿到用户标识来做唯一性判断。

## 用户标识不能从客户端获取

聊到用户登录，业务代码非常容易犯错的一个地方是，使用了客户端传给服务端的用户 ID，类似这样：

 复制代码

```
1 @GetMapping("wrong")
2 public String wrong(@RequestParam("userId") Long userId) {
3     return "当前UserId: " + userId;
4 }
```

你可能觉得没人会这么干，但我就真实遇到过：**一个大项目因为服务端直接使用了客户端传过来的用户标识，导致了安全问题。**


犯类似低级错误的原因，有三个：

开发同学没有正确认识接口或服务面向的用户。如果接口面向内部服务，由服务调用方传入用户 ID 没什么不合理，但是这样的接口不能直接开放给客户端或 H5 使用。

在测试阶段为了方便测试调试，我们通常会实现一些无需登录即可使用的接口，直接使用客户端传过来的用户标识，却在线之前忘记删除类似的超级接口。

一个大型网站前端可能由不同的模块构成，不一定是一个系统，而用户登录状态可能也没有打通。有些时候，我们图简单可能会在 URL 中直接传用户 ID，以实现通过前端传值来打通用户登录状态。

如果你的接口直面用户（比如给客户端或 H5 页面调用），那么一定需要用户先登录才能使用。登录后用户标识保存在服务端，接口需要从服务端（比如 Session 中）获取。这里有段代码演示了一个最简单的登录操作，登录后在 Session 中设置了当前用户的标识：

 复制代码

```
1 @GetMapping("login")
2 public long login(@RequestParam("username") String username, @RequestParam("password") String password) {
3     if (username.equals("admin") && password.equals("admin")) {
```

```
4         session.setAttribute("currentUser", 1L);
5         return 1L;
6     }
7     return 0L;
8 }
```

这里，我再分享一个 Spring Web 的小技巧。

如果希望每一个需要登录的方法，都从 Session 中获得当前用户标识，并进行一些后续处理的话，我们没有必要在每一个方法内都复制粘贴相同的获取用户身份的逻辑，可以定义一个自定义注解 `@LoginRequired` 到 `userId` 参数上，然后通过 `HandlerMethodArgumentResolver` 自动实现参数的组装：

```
1 @GetMapping("right")
2 public String right(@LoginRequired Long userId) {
3     return "当前UserId: " + userId;
4 }
```

 复制代码

`@LoginRequired` 本身并无特殊，只是一个自定义注解：

```
1 @Retention(RetentionPolicy.RUNTIME)
2 @Target(ElementType.PARAMETER)
3 @Documented
4 public @interface LoginRequired {
5     String sessionKey() default "currentUser";
6 }
```

 复制代码

魔法来自 `HandlerMethodArgumentResolver`。我们自定义了一个实现类 `LoginRequiredArgumentResolver`，实现了 `HandlerMethodArgumentResolver` 接口的 2 个方法：

`supportsParameter` 方法判断当参数上有 `@LoginRequired` 注解时，再做自定义参数解析的处理；

resolveArgument 方法用来实现解析逻辑本身。在这里，我们尝试从 Session 中获取当前用户的标识，如果无法获取到的话提示非法调用的错误，如果获取到则返回 userId。这样一来，Controller 中的 userId 参数就可以自动赋值了。

 复制代码

```
1 @Slf4j
2 public class LoginRequiredArgumentResolver implements HandlerMethodArgumentRes
3     //解析哪些参数
4     @Override
5     public boolean supportsParameter(MethodParameter methodParameter) {
6         //匹配参数上具有@loginRequired注解的参数
7         return methodParameter.hasParameterAnnotation(LoginRequired.class);
8     }
9
10
11     @Override
12     public Object resolveArgument(MethodParameter methodParameter, ModelAndView
13         //从参数上获得注解
14         LoginRequired loginRequired = methodParameter.getParameterAnnotation(L
15         //根据注解中的Session Key, 从Session中查询用户信息
16         Object object = nativeWebRequest.getAttribute(loginRequired.sessionKey
17         if (object == null) {
18             log.error("接口 {} 非法调用!", methodParameter.getMethod().toString(
19             throw new RuntimeException("请先登录!");
20         }
21         return object;
22     }
23 }
```

当然，我们要实现 WebMvcConfigurer 接口的 addArgumentResolvers 方法，来增加这个自定义的处理器 LoginRequiredArgumentResolver：

 复制代码

```
1 SpringBootApplication
2 public class CommonMistakesApplication implements WebMvcConfigurer {
3     ...
4     @Override
5     public void addArgumentResolvers(List<HandlerMethodArgumentResolver> resolu
6         resolvers.add(new LoginRequiredArgumentResolver());
7     }
8 }
```

测试发现，经过这样的实现，登录后所有需要登录的方法都可以一键通过加 @LoginRequired 注解来拿到用户标识，方便且安全：

← → ↻ 🏠 ⓘ localhost:45678/trustclientuserid/right

## 当前用户Id: 1

### 重点回顾

今天，我就“任何客户端的东西都不可信任”这个结论，和你讲解了一些有代表性的错误。

第一，客户端的计算不可信。虽然目前很多项目的前端都是富前端，会做大量的逻辑计算，无需访问服务端接口就可以顺畅完成各种功能，但来自客户端的计算结果不能直接信任。最终在进行业务操作时，客户端只能扮演信息收集的角色，虽然可以将诸如价格等信息传给服务端，但只能用于校对比较，最终要以服务端的计算结果为准。

第二，所有来自客户端的参数都需要校验判断合法性。即使我们知道用户是在一个下拉列表选择数据，即使我们知道用户通过网页正常操作不可能提交不合法的值，服务端也应该进行参数校验，防止非法用户绕过浏览器 UI 页面通过工具直接向服务端提交参数。

第三，除了请求 Body 中的信息，请求头里的任何信息同样不能信任。我们要知道，来自请求头的 IP、Referer 和 Cookie 都有被篡改的可能性，相关数据只能用来参考和记录，不能用作重要业务逻辑。

第四，如果接口面向外部用户，那么一定不能出现用户标识这样的参数，当前用户的标识一定来自服务端，只有经过身份认证后的用户才会在服务端留下标识。如果你的接口现在面向内部其他服务，那么也要千万小心这样的接口只能内部使用，还可能需要进一步考虑服务端调用方的授权问题。

安全问题是木桶效应，整个系统的安全等级取决于安全性最薄弱的那个模块。在写业务代码的时候，要从我做起，建立最基本的安全意识，从源头杜绝低级安全问题。

今天用到的代码，我都放在了 GitHub 上，你可以点击 [这个链接](#) 查看。

## 思考与讨论

1. 在讲述用户标识不能从客户端获取这个要点的时候，我提到开发同学可能会因为用户信息未打通而通过前端来传用户 ID。那我们有什么好办法，来打通不同的系统甚至不同网站的用户标识吗？
2. 还有一类和客户端数据相关的漏洞非常重要，那就是 URL 地址中的数据。在把匿名用户重定向到登录页面的时候，我们一般会带上 redirectUrl，这样用户登录后可以快速返回之前的页面。黑客可能会伪造一个活动链接，由真实的网站 + 钓鱼的 redirectUrl 构成，发邮件诱导用户进行登录。用户登录时访问的其实是真的网站，所以不容易察觉到 redirectUrl 是钓鱼网站，登录后却来到了钓鱼网站，用户可能会不知不觉就把重要信息泄露了。这种安全问题，我们叫做开放重定向问题。你觉得，从代码层面应该怎么预防开放重定向问题呢？

你还遇到过因为信任 HTTP 请求中客户端传给服务端的信息导致的安全问题吗？我是朱晔，欢迎在评论区与我留言分享你的想法，也欢迎你把今天的内容分享给你的朋友或同事，一起交流。

### 课程预告

# 6月-7月课表抢先看

## 充 ¥500 得 ¥580

赠「¥ 118 月球主题 AR 笔记本」



【点击】图片，立即查看 >>>

上一篇 26 | 数据存储: NoSQL与RDBMS如何取长补短、相辅相成?

下一篇 28 | 安全兜底: 涉及钱时, 必须考虑防刷、限量和防重

## 精选留言 (7)

写留言



**ddosyang**

2020-05-24

第一个订单例子的right方法, 第六行是不是应该改为if (!order.getItemPrice().equals(item.getItemPrice()))? 因为是想判断不等于的情况, 所以这里是不是漏了一个叹号?

作者回复: 是的, 我改一下



1



**Darren**

2020-05-19

第一个问题: 统一登陆获取x-toekn (jwt) 统一鉴权 (解析x-toekn), 前端请求过网关, 网关处理x-toekn, 根据x-toekn解析用户ID, 用户名等, 存放到header中, 同时也保留x-toekn, 后面的微服务直接获取即可。全局base包, 里面定义header中的userid, username, x-toekn等信息, 这样既是该服务调用别的服务, 别的服务也涉及x-toekn也是可以的。...

展开

作者回复: 嗯是jwt 笔误



1



**汝林外史**

2020-05-20

1. 就是用面试中经常问的单点登录实现。说白了就是把token专门放在一个地方存着, 再给客户端个凭证, 等客户端需要校验是否登录的时候就用这个凭证去存token的服务器校验下, 通过了就直接登录, 不通过就跳转到登录页。

2. 可以校验下redirectUrl吧

展开



**梦倚栏杆**

2020-05-19

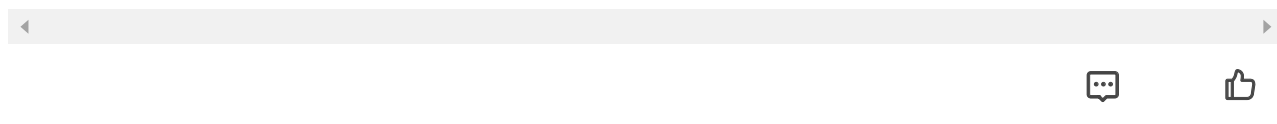


1.是统一登录

2.老师介绍的这些是说端上的内容不可信，那后层服务呢？假如我问有一个统一的网关，可以确认用户登录，那么我们应该相信网关吗？如果相信，是不是强依赖网关了，网关有问题，服务就有问题。但是如果不相信，网关就起不到作用了

展开 ▾

作者回复: 如果网关做了正确的身份认证那么可以相信，一般把用户Token转换为用户ID的这个工作就是由网关来做的，网关后面的微服务无需再处理身份认证的工作



**那一刻**

2020-05-19

讨论题，谈谈我的不成熟想法。

- 1.不同系统用户标示，可以采用设备ID。或者采用统一的登陆系统来标示用户。
- 2.开放重定向问题，首先，不能采用传来的url作为redirect的base url。其次，redirect url写全包含host。不知还有没有其它防御手段？

展开 ▾



**fly12580**

2020-05-19

还可以对请求参数进行加密，在服务端进行解析判断。加强安全性。



**Demon.Lee**

2020-05-19

1. 未想到特别方便的方法，很快就能打通
2. 查询资料，一般对redirectUrl进行域名校验，并先跳转到一个统一的页面，并提示用户会离开当前网站，类似的“知乎”，“简书”都是这么设计的。

作者回复: 1、可以使用JWT Token进行打通，或走SSO体系

2、抛砖引玉：

- 1)、固定重定向的目标URL；
- 2)、可采用编号方式指定重定向的目标URL；
- 3)、合理充分的校验校验跳转的目标地址，非己方地址时告知用户跳转风险；

