美团面试：接口被恶意狂刷，怎么办？

Original 剽悍一小兔 java小白翻身 2023-05-30 07:15 Posted on 江苏

如果Java接口被恶意狂刷，我们一般可以采取以下措施：

用TimeStamp （兵不厌诈）

比如给客户端提供一个timestamp参数，值是13位的毫秒级时间戳，可以在第12位或者13位做一个校验位，通过一定的算法给其他12位的值做一个校验。

举例：现在实际时间是 1684059940123，我通过前12位算出来校验位是9，参数则把最后一位改成9，即1684059940129，值传到服务端后通过前十二位也可以算出来值，来判断这个时间戳是不是合法的。

根据前面12位计算校验位，可以使用以下代码实现：

```java
/**
 * 计算timestamp的校验位
 */
private int calculateCheckDigit(long timestamp) {
    int sum = 0;
    for (int i = 0; i < TIMESTAMP_DIGITS - 1; i++) {
        char c = String.valueOf(timestamp).charAt(i);
        sum += Integer.parseInt(String.valueOf(c));
    }
    return sum % 10;
}
```

在上述代码中，calculateCheckDigit方法接收一个13位毫秒级时间戳作为参数，返回该时间戳的校验位。具体地，该方法会循环遍历时间戳的前12位，并将这些数字相加。最后，计算总和的个位数并返回。

例如：如果时间戳为1684059940123，则该方法计算出来的校验位为9。

但是如果黑客翻客户端代码，就知道校验位的情况了。因为这个校验逻辑客户端必然也有一份。如果客户端代码是公开的，那么攻击者可以通过研究客户端代码得到校验位计算的方式。这样一来，攻击者就有可能伪造出带有正确校验位的timestamp参数，从而绕过Java接口的限流和安全机制。

因此，该方案主要适用于需要简单防范一些低强度攻击的场景，例如防范垃圾请求或非法爬虫等。对于高强度攻击，建议采取更为复杂的验证策略，例如使用OAuth2授权、IP白名单、签名算法等。同时，建议客户端和服务端在通信过程中使用HTTPS协议进行加密，防止数据被窃听或篡改。

加强安全认证

```java
@RequestMapping("/api/login")
public String login(@RequestParam("username") String username, @RequestParam("password") S
    if(!checkUser(username, password)){
        return "用户名或密码错误";
    }
    String token = getToken();
    saveToken(token);
    return token;
}

private boolean checkUser(String username, String password){
    //校验用户是否合法
}

private String getToken(){
    //生成token
}

private void saveToken(String token){
    //保存token
}
```

在上述代码中，当用户调用login接口时，需要提供用户名和密码。此时会进行用户校验，若校验失败则返回错误信息，否则生成token并保存，最终返回给用户。

生成Token的作用是为了在接口请求时验证用户身份。具体来说，当用户第一次登录系统后，该接口可以根据用户信息生成一个Token字符串，并将其保存至服务端或客户端。当此用户访问其他需要鉴权的接口时，需要在请求头中带上这个Token字符串，以便服务器进行身份验证。

对于Java接口被恶意狂刷问题，Token的作用是防止非法请求。由于Token是由服务端生成的，攻击方无法自己生成有效的Token，因此只有拥有合法Token的用户才能成功调用相关接口。

关于Token的验证，可以通过拦截器实现。拦截器可以在接口调用前检查请求头中是否包含合法的Token，并验证Token是否过期、是否被篡改等。如果Token验证失败，则返回错误信息并拦截该请求。

下面是生成Token和使用拦截器的示例代码：

```
// 生成Token
private String generateToken(User user) {
    // 基于用户名、密码、时间戳等信息生成Token字符串
}

// 鉴权拦截器
public class AuthInterceptor extends HandlerInterceptorAdapter {

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response,
                             Object handler) throws Exception {
        String token = request.getHeader("Authorization");
        if (token == null || !checkToken(token)) {
            response.setStatus(HttpStatus.UNAUTHORIZED.value());
            return false;
        }
        return true;
    }

    private boolean checkToken(String token) {
        // 验证Token是否合法，是否过期等
    }
}
```

在上述代码中，generateToken方法用于生成Token字符串，并保存至服务端或客户端。AuthInterceptor类是拦截器类，用于检查请求头中的Token是否合法。如果Token验证失败，则返回401错误码并拦截该请求。需要注意的是，在使用该拦截器时，需要将其注册到Spring MVC框架中。

```
@Configuration
public class WebMvcConfig implements WebMvcConfigurer {

    @Autowired
    private AuthInterceptor authInterceptor;

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(authInterceptor).addPathPatterns("/api/**");
    }
}
```

以上代码是将AuthInterceptor拦截器注册到Spring MVC框架中，使其生效。其中，"/api/**"表示拦截所有以"/api"开头的接口。

IP封禁

```java
public class IpFilter extends OncePerRequestFilter {
    private static final Set<String> IP_SET = new HashSet<>();

    static {
        IP_SET.add("192.168.1.100");
        IP_SET.add("127.0.0.1");
        //添加其他需要封禁的IP
    }

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse respon
                                    FilterChain filterChain) throws ServletException, IOEx
        String ipAddress = request.getRemoteAddr();
        if(IP_SET.contains(ipAddress)){
            response.setStatus(HttpStatus.FORBIDDEN.value());
            return;
        }

        filterChain.doFilter(request, response);
    }
}
```

在上述代码中，通过IpFilter过滤器来阻止特定的IP地址访问接口。其中，IP_SET为需要封禁的IP地址集合。

接口限流

```java
public class RateLimitInterceptor extends HandlerInterceptorAdapter {
    private static final int DEFAULT_RATE_LIMIT_COUNT = 100; //默认每秒允许100次请求
    private static final int DEFAULT_RATE_LIMIT_TIME_WINDOW = 1000; //默认时间窗口为1秒
    private Map<String, Long> requestCountMap = new ConcurrentHashMap<>();
    private int rateLimitTimeWindow;
    private int rateLimitCount;

    public RateLimitInterceptor(){
        this(DEFAULT_RATE_LIMIT_TIME_WINDOW, DEFAULT_RATE_LIMIT_COUNT);
    }

    public RateLimitInterceptor(int timeWindow, int limitCount){
        this.rateLimitTimeWindow = timeWindow;
        this.rateLimitCount = limitCount;
    }

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Obj
            throws Exception {
        String key = request.getRemoteAddr() + ":" + request.getRequestURI();
        long now = System.currentTimeMillis();
        if(requestCountMap.containsKey(key)){
            long last = requestCountMap.get(key);
            if(now - last < rateLimitTimeWindow){
                if(requestCountMap.get(key) >= rateLimitCount){
                    response.setStatus(HttpStatus.TOO_MANY_REQUESTS.value());
                    return false;
                }
                requestCountMap.put(key, requestCountMap.get(key) + 1);
            }else{
                requestCountMap.put(key, now);
            }
        }else{
            requestCountMap.put(key, now);
        }
        return true;
    }
}
```

在上述代码中，通过RateLimitInterceptor拦截器实现接口限流功能。默认情况下，每秒钟最多允许100次请求。如果当秒内请求数超过限制，则返回状态码429。

日志监控

```
public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
    throws IOException, ServletException {
    HttpServletRequest httpRequest = (HttpServletRequest) request;
    HttpServletResponse httpResponse = (HttpServletResponse) response;
    String requestURI = httpRequest.getRequestURI();

    try {
        log.info("Request Received. URI: {}", requestURI);
        chain.doFilter(request, response);
    } catch(Exception e) {
        log.error("Exception occurred while processing request. URI: {}", requestURI, e);
        throw e;
    } finally {
        log.info("Request Completed. URI: {} Response Status: {}", requestURI, httpRespons
    }
}
```

在上述代码中，通过Filter过滤器来实现日志监控。当请求进入时记录请求URI，当请求结束时记录响应状态码，如此可及时发现异常情况。

升级硬件设备

如果服务器无法承受恶意攻击，可以通过升级硬件设备来增加服务器的承载能力。例如，可以增加CPU或内存等硬件资源，降低服务器的响应时间。

通知相关部门

当Java接口被恶意狂刷时，及时通知相关管理人员或安全团队是非常重要的。他们可以采取更加有效的措施，如封禁IP地址、加强认证机制等，从而保障接口的安全。以下是一些示例代码：

```
//发送邮件通知相关部门
public void sendNotificationEmail(String subject, String content, String... recipients) {
    SimpleMailMessage message = new SimpleMailMessage();
    message.setFrom("sender@example.com");
    message.setTo(recipients);
    message.setSubject(subject);
    message.setText(content);

    mailSender.send(message);
}

//发送短信通知相关部门
public void sendNotificationSMS(String phoneNumber, String content) {
    //调用第三方短信API发送短信
}
```

总结

1.大部分情况，token校验+拦截器已然足够，敏感接口再加限流，这也是大部分企业的做法。

2.简单场景用TimeStamp就行了，对于大部分机器人恶意访问，他是不可能去看你代码的。

3.尽量不要封禁IP，防止误伤，因为很多小区住户往往是局域网，很多人就是共用一个IP的。

4.更狠的方法是，模仿那种"请选择图片中的汽车"的做法，反向薅羊毛，嗯，大家应该明白其中的意思。