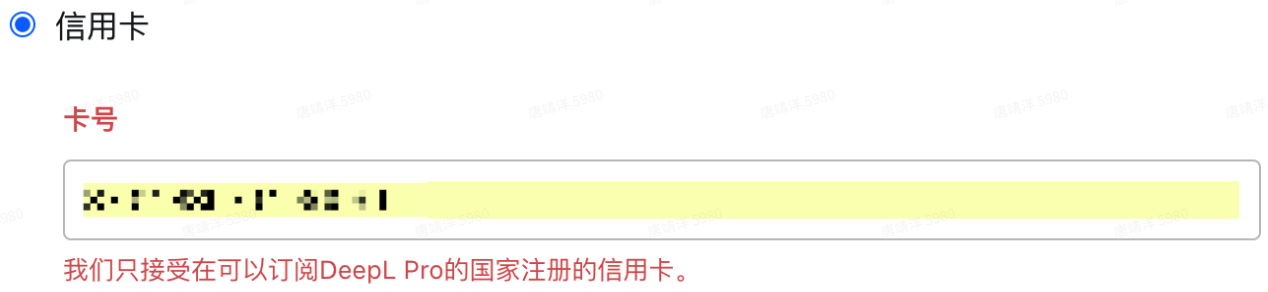
一个有意思的API安全策略

背景

近期有批量翻译的需要，最终选中DeepL的翻译服务，然而他们的付费API直到我付款的时候才发现并不支持在中国注册的信用卡。



但是经过同行对比，又只看中了DeepL的翻译质量，于是F12看了看官网的翻译请求，看看能不能另辟蹊径。

初探

F12打开后意外发现，翻译的接口参数意外的简单，且每个参数都有明确的语义，没有任何的混淆。

```
{
  "jsonrpc": "2.0",
  "method": "LMT_handle_jobs",
  "params": {
    "jobs": [
      {
        "kind": "default",
        "sentences": [{ "text": "hello", "id": 0, "prefix": "" }],
        "raw_en_context_before": [],
        "raw_en_context_after": [],
        "preferred_num_beams": 4,
        "quality": "fast"
      }
    ],
    "lang": {
      "preference": {
        "weight": {
          "DE": 0.29615,
          "EN": 13.81927,
          "ES": 0.40291,
          "FR": 0.2464,
          "IT": 0.79363,
          "JA": 0.09171,
          "NL": 0.21536,
          "PL": 0.15462,
          "PT": 0.24719,
          "RU": 0.04946,
          "ZH": 0.34183,
          "CS": 0.16552,
          "DA": 0.15559,
          "ET": 0.22984,
          "FI": 0.19149,
```

```
          "HU": 0.09301,
          "LT": 0.06693,
          "LV": 0.05934,
          "RO": 0.10224,
          "SK": 0.14867,
          "SL": 0.09171,
          "SV": 0.16133,
          "TR": 0.07354,
          "ID": 0.07074,
          "BG": 0.03518,
          "EL": 0.03417,
          "UK": 0.05292
        },
        "default": "default"
      },
      "source_lang_user_selected": "auto",
      "target_lang": "ZH"
    },
    "priority": -1,
    "commonJobParams": {
      "mode": "translate",
      "browserType": 1,
      "formality": null
    },
    "timestamp": 1668913990555
  },
  "id": 83110013
}

```


于是迅速对接好接口，本以为事情已经到此为止的时候，接口的响应告诉我没那么简单，429状态码应该是被判定为非法请求了。

```
2022/11/20 11:25:49 http status is not ok: 429 Too Many Requests
```

回顾了下上述的接口，发现每次请求只有两个参数是动态的：

1. `timestamp`，实时的unixtime
2. `id`，可疑，尚不知道具体的生成算法

既然这个`id`是请求发起方生成的，那么他的算法一定包含在某段js代码里，然而看了下js代码都是经过混淆的，想要读明白具体啥意思，ROI极低。

进展

因为`id`的具体生成方式难以知晓，即将要放弃的时候，我抱着侥幸的心理上网搜了搜，因为事实上的侵权，有关api的分析貌似被DeepL发了律师函也都被删除了（声明：本文章只用作本次周会分享交流）：

不过通过搜索我获取到一个关键线索：**windows的客户端**（~~破解还得是靠windows~~）

突破

于是打开尘封已久的windows电脑，下载好DeepL的客户端，抓了下包发现，接口参数更为简单了！结构如下：

```
{
  "jsonrpc": "2.0",
  "method": "LMT_handle_texts",
  "params": {
    "lang": {
      "source_lang_user_selected": "auto",
      "target_lang": "ZH"
    },
    "commonJobParams": {},
    "timestamp": 1668914390431
  },
  "id": 33216210
}
```

每次请求依然是只有两个动态参数：

1. `timestamp`
2. `id`

和之前分析的没错，一切的玄机，尽在`id`这个参数。

从安装目录可以看到DeepL的win客户端是C#写的，逆向起来一般比较容易，这个客户端加固措施简陋，很顺利就看到了源代码。。

参数：`id`

既然秘密都在`id`上，我们就直奔主题：

```
this.nextId = (long)Math.Round(randomSource.NextDouble() * 10000.0) * 10000L;
```

实际上，`id`这个参数没啥高大上的生成算法，无非就是一个[0, 1)的随机浮点数*10000后取整，再*10000即可。

没错，暗藏玄机的`id`**本质上也就是个随机数。**。于是改了下自己的`id`生成策略，再次发起了请求，正以为大功告成的时候，接口的响应还是报429。为了避免是IP被ban了，更换代理也无果。

参数：`timestamp`

再次回到源码才发现，秘密回到了`timestamp`这个人畜无害的参数上，本以为它是当前的时间戳，但是没想到是经过二次计算的：`当前毫秒时间戳 - 当前毫秒时间戳 % 翻译文本含字母i的数量 + 翻译文本含字母i的数量`

> 后来想想，仅靠id这个随机数来做接口防护确实也不太可能

生成timestamp：

```
// DeepL.Rpc.TextTranslation.TextTranslationRules
// Token: 0x0600004C RID: 76 RVA: 0x000026F4 File Offset: 0x000008F4
public long GenerateTimestamp(TextDto[] texts)
{
    long num = this.dateTimeOffsetSource.UtcNow.ToUnixTimeMilliseconds();
    long num2 = texts.Aggregate(1L, (long i, TextDto t) => i + (long)this.iCounter.CountIs(t.Text));
    return num - num % num2 + num2;
}
```

计算文本包含字母`i`（ascii: 105）的数量:

```
public int CountIs(string text)
{
    return Encoding.ASCII.GetBytes(text).Count((byte c) => c == 105);
}
```

于是改了下自己的 `timestamp` 生成策略，再次发起了请求，再次以为大功告成的时候，接口的响应还是报429。

请求：`body`

再次回到源码，谁又能想到，在请求真正发起的时候，http请求的body又被做了一次手脚呢？并且表面只是一个随机数的 `id` 参数，它却起着关键性的作用呢？

从下面的代码我们可以看到，http body中的json，会依据 `id` 的实际值，做一些调整：

原始compact json中的一个子字符串: `"method":"`

- 如果 (id + 3) % 13 == 0 || (id + 5) % 29 == 0，则会调整为: `"method" : "`
- 否则调整为： `"method": "`

```
// DeepL.Rpc.TextTranslation.TextTranslationRules
// Token: 0x06000047 RID: 71 RVA: 0x00002618 File Offset: 0x00000818
public string AdjustJsonContent(long id, string jsonContent)
{
    return jsonContent.Replace("\"method\":\"", ((id + 3L) % 13L == 0L || (id + 5L) % 29L == 0L) ? "\"method\" : \"" :
      "\"method\": \"");
}
```

回头看看F12，发现确确实实method这个key后面多了一个空格：

也就是说，这个最为关键的特称，在我最开始格式化json的那一刻就被抹除了

感想

至此，接口已经能够正常翻译文本了，不过今天回头想想还是颇有感想：

1. 整个过程表面上被攻击的是api，实际上被攻击的是心态，下面是几记关键重拳：
    - 费尽心思发现关键参数 `id` 仅仅是一个随机数的时候
    - 发现寻常参数 `timestamp` 实际上却并不寻常的时候
    - 发现参数的生成方式都正确但请求body被暗改的时候
    - 发现仅仅是一个随机数的 `id` 却起着关键作用的时候（虾仁猪心）
2. api防护，除了鉴权，签名，参数混淆等常规手段，尝试从人类的心理来预防或许也是一个绝妙的思路（有点类似蜜罐）
3. api的安全，在多个端之间存在木桶效应，~~谨防猪队友~~