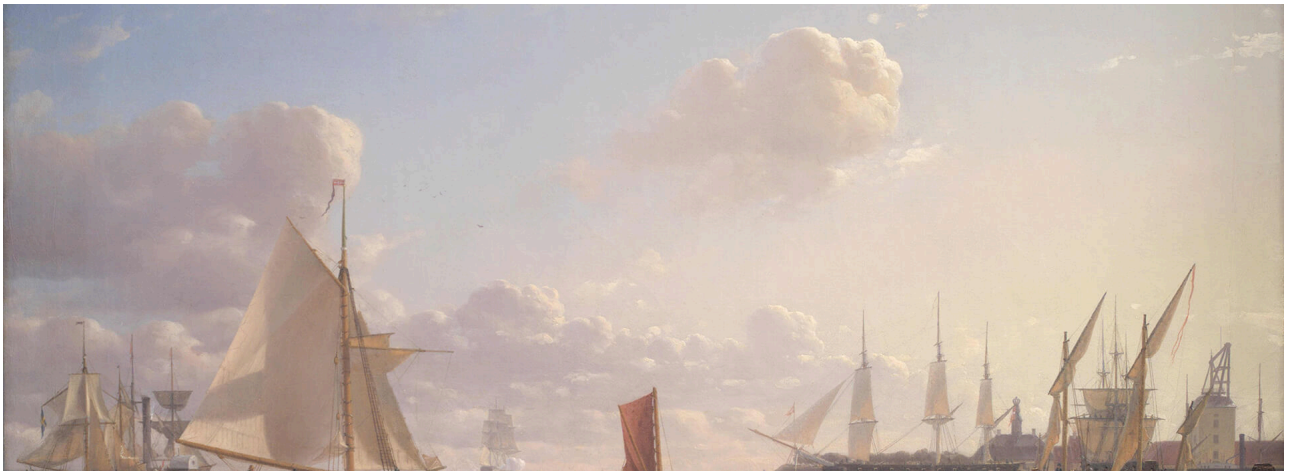

基础 2–无限 Debugger 调试与处理

当我们稍微学习了一些 js 基础知识，加上一些 Devtools 之后，就可以轻松的搞定

1 入门 Hook 基础

在讲这个之前，我们来入门一个 hook，不为别的，为了能够更好的让我们理解今天的课程

至于 hook 全方案，我们以后再讲

全方案链接：

卢振南同学的宝藏教程–扣代码补环境 (dingtalk.com)

1.1 hook 的定义

hook 八股文还是大家自己百度。我说一下我自己的理解

hook，就是钩子，相当于把一个方法勾着，勾住之后呢，我们给原来的函数/功能 加层壳

壳的最后要执行之前的逻辑，剩下的壳子里功能，我们按照自己的想法来定制

```
function a(){
    return arguments[0]
}
```

有这样的一个函数 a,这个 a 假设我们是无法修改的，也就是不能动，但是，我想看到传入 a 里面的第二个参数

那么怎么办呢?

```
  _a = a   // 这一步叫保留原始函数
  a = function(){   // 这一步叫重写函数
      console.log(arguments[1])   //  实现我们想要功能和逻辑的地方
      return _a(arguments)        //  保留原始的功能，让原来的内容可以正常执行
  }
```

概念区分：（广义，约定，习惯）

1.2 hook 与 重写（debugger 三元素）

重写： 改变掉原先函数的逻辑，让功能变化

hook：让之前函数的功能不发生变化，只不过让我们随心所欲的加入一些东西，去控制它的情况

首先，我们要知道，在浏览器实现 debugger 的方法有哪些

1. debugger 关键词（最经典的了，基础课大家应该太熟悉了，我们经常用这个关键词做调试）
2. eval('debugger') 原理跟 1 类似，只不过是在虚拟机里面执行 debugger 的方法
3. Function('debugger')() 及其变种 原理跟 2 类似，只不过是在虚拟机里面执行匿名函数，匿名函数里有 debugger 的方法

这些 debugger 方法，是实现 debugger 的基础，可以理解为是三元素，基于三种元素，可以形成多种多

这些 debugger 方法，是实现 debugger 的基础，可以理解为是三元素。基于三种元素，可以形成多种多样的玩法

在我们接触无限 debugger 之前，要先了解一点

1. 无限 debugger 一定是有限 debugger，不信邪的执行一下下面的代码：

   ```
   while(1){debugger;}
   ```

   纯文本

   你看浏览器扎不扎你就完了

   只不过这个执行次数多到我们本身靠手点难以接受罢了

2. 三元素之间还可以相互组合，并且套任何的方式。虽然最终实现的原理是这个，但是中间的过程可能会非常复杂
3. 解决无限 debugger 名词解释应该为：在没有 debugger 干扰的情况下调试，而不是：放弃所有的 debugger 调试

   （也就是说，我们自己的调试还得能正常使用）

2 debugger 三元素的组合反调试示范

那么接下来， 我就将上面的三元素进行组合，产生各种无限 debugger（不一定全，但是大家感受一下即可）

2.1 定时器反调试

```
setInterval('debugger', 500)
```

纯文本

这种是利用定时器的方法，去不断间隔的创造宏任务队列，用来干扰我们的调试

2.1.1 解决方案

a. 在定时器启动之前重写 setInterval

b. Never pause here

c. AutoResponse 直接把干扰点干掉

2.2 script 标签生成反调试

1. html 页面生产时，直接自动生成几千个 script 标签，标签里只写一个 debugger

   ### 2.2.1 解决方案

AutoResponse 直接把干扰点干掉

2.2.2 拓展 1：页面动态

反爬方对抗方案：每次页面都是动态的，替换了校验不通过

攻方对抗方案： 1. 找到动态点，动态会坍缩成静态

a. 校验不通过又如何，只要 js 能调试，就代表着，这轮代码我调了。总不能每一次都返回全新的 script 吧?

2.2.3 拓展 2：debugger 生成方式随机且手工修改

反爬方对抗方案：我的 debugger 生成方式都是随机的，三元素我各种拼接组合，而且每天还手动修改

攻方对抗方案： 编译浏览器，让 script 的 debugger 直接不触发，只出发我手动打断点的位置

.....

2.3 循环反调试

纯文本

```
for (let i=0;i<=5000;i++){
   Function('debugger')()  // 原型链 获取Function
   eval('debugger')
   debugger;
}
```

2.3.1 解决方案

a. 干掉这个循环进入的函数

b. Never pause here

c. AutoResponse 直接把干扰点干掉

2.3.2 拓展 1：Never pause here 对抗

纯文本

```
   j = '';
   for (let i=0;i<=5000;i++){
       j += ';'
       Function('debugger' + j)()
       eval('debugger' + j)
       debugger;
    }
```

解决方案：

a. 干掉这个循环进入的函数

b. AutoResponse 直接把干扰点干掉

2.3.3 拓展 2：AutoResponse 对抗

'debugger' 字符串用业务代码 + 一大堆乱七八糟的方式算出来，最后给个返回值 debugger

解决方案：

对抗这个函数，把返回值干掉

.....

2.4 添加 script 标签插入 debugger 反调试

```
// 外面再套循环体，懒得写了
cont = document.body
var newScript = document.createElement('script');
newScript.type = 'text/javascript';
newScript.innerHTML = 'debugger';
cont.appendChild(newScript);
```

2.4.1 解决方案

document.createElement 里面把 script 重写

```
var _b = new Blob(['debugger'], {type: 'application/javascript'})
var _w = new Worker(URL.createObjectURL(_b))
blob_ = Blob.prototype.constructor
Blob.prototype.constructor = function(){
debugger;
if(Array.isArray(arguments[0]) && arguments[0][0].indexOf('debugger') != -1){
    arguments[0][0] = ''

}
return new blob_(arguments)

}
Blob = Blob.prototype.constructor
```

2.5 关于无限 debugger 实现的拓展

以上所有方法都可以嵌套使用，我只是表示一部分逻辑。但是无论怎么嵌套，处理的方案都是一致的

2.5.1 优先尝试 Never pause here

最方便快捷，但是最卡，也最容出问题

2.5.2 次优先尝试重写调用函数

如：

```
Function = function(){}
setInterval = function(){}
```

缺陷：容易破坏业务逻辑，导致控制流变化

2.5.3 使用 AutoResponse/mapping/overrides 替换

缺陷：操作稍微有一点点的麻烦，对动态情况的支持不太好，也可能会改变控制流走向

2.5.4 如果都不行.....

1. 万一以上三个都难受了怎么办？ 别逆向了，反调试都那么难了，那加密不得难上天啊。放弃吧，嗷～

   【自定义浏览器？重新编译浏览器？魔改 debugger 逻辑？.....】

```
// 为什么这么写，hook eval
 _eval = eval;
eval= function(){
    if (arguments[0].indexOf('debugger') === -1)
    return _eval(arguments[0])
}

// 为什么这么写，原型链，基础
A = function(){};
A.prototype.bbb =  function(){console.log('11111')}
aaa = new A();
A.prototype.bbb = function(){console.log('55555')}
aaa.bbb()



_appendChild = Node.prototype.appendChild
Node.prototype.appendChild = function(){
    if (arguments[0].innerHTML && arguments[0].innerHTML.indexOf('debugger') != -1){
         arguments[0].innerHTML = ''
    }
    return _appendChild.apply(this, arguments)
}
```

然后把其他的 script 里的 debugger 用 autoResponse/overrides/mapping 替换掉

```
// 思考 , Function 重写为什么不行?
_Function = Function
Function = function(){
    if (arguments[0].indexOf('debugger') != -1){
        return _Function('')
    }
    return _Function(arguments[0])
}
// 学过原型链的大家应该都清楚了
```

已知：

1. Function.constructor = Function
2. 所有的函数定义，实际上都是 new Function
3. 也就是说，函数实际上是 Function 的实例化对象
4. 那么函数的 constructor，实际上就是 Function.prototype.constructor
5. 所以 Function.prototype.constructor = Function
6. 所以只要修改 Function.prototype.constructor，就可以实现 hook 自定义函数的 constructor 目的

所以应该这样写：

```
_Function = Function
Function.prototype.constructor = function(){
    if (arguments[0].indexOf('debugger') != -1){
            arguments[0] = arguments[0].replaceAll('debugger','')
        }
    return _Function(arguments[0])
}
```