JavaScript逆向调试记 —— defcon threefactooorx writeup

原创

落沐萧萧 于 2021-05-04 14:48:57 发布 167 收藏
文章标签： java js python javascript web

defcon 29就这一道Web题目，说实话也没学到啥东西，唯一学到的就是勿钻牛角尖，及时调整策略。

此题严格来说算一道逆向题，只不过逆向的目标是混淆过JavaScript，我方法就是硬逆，等过几天看看其他人writeup，也许会有更简单的方法。先不讨论这些方法了，仅说下我自己的流水账。

PS. 本文没有做过题目的同学也可以看看。

0x01 确定考点

拿到题目：

> This is the end of phishing. The Order of the Overflow is introducing the ultimate authentication factor, the most important one, the final one. To help the web transition to this new era of security, we are introducing a 3FA tool for testing your webpages completely isolated on our admin's browser.
>
> **http://threefactooorx.challenges.ooo:4017**
>
> Files:
>
> 3factooorx.crx b40cabadcdbf1d0a8868121d184fcb9d5355c688045dc5a2e91fe870e846ff1d

首先看一下给出的URL，里面就是一个上传HTML的页面，上传一个HTML后，有个Bot会访问这个页面，并把截图发回来。再就是，给了一个crx格式的附件，说明这道题和浏览器插件有关。

下载这个crx，解压后，查看最重要的menifest.json，这是浏览器插件的配置文件：

```
{
  "manifest_version": 2,
  "name": "3FACTOOORX",
  "description": "description",
  "version": "0.0.1",
  "icons": {
    "64": "icons/icon.png"
  },
  "background": {
    "scripts": [
      "background_script.js"
    ]
  },
  "content_scripts": [
    {
      "matches": [
        "<all_urls>"
      ],
      "run_at": "document_start",
      "js": [
        "content_script.js"
      ]
    }
  ],
  "page_action": {
    "default_icon": {
      "64": "icons/icon.png"
    },
    "default_popup": "pageAction/index.html",
    "default_title": "3FACTOOORX"
  }
}
```

两个重要的文件：

background_script.js 这个脚本会运行在插件后台

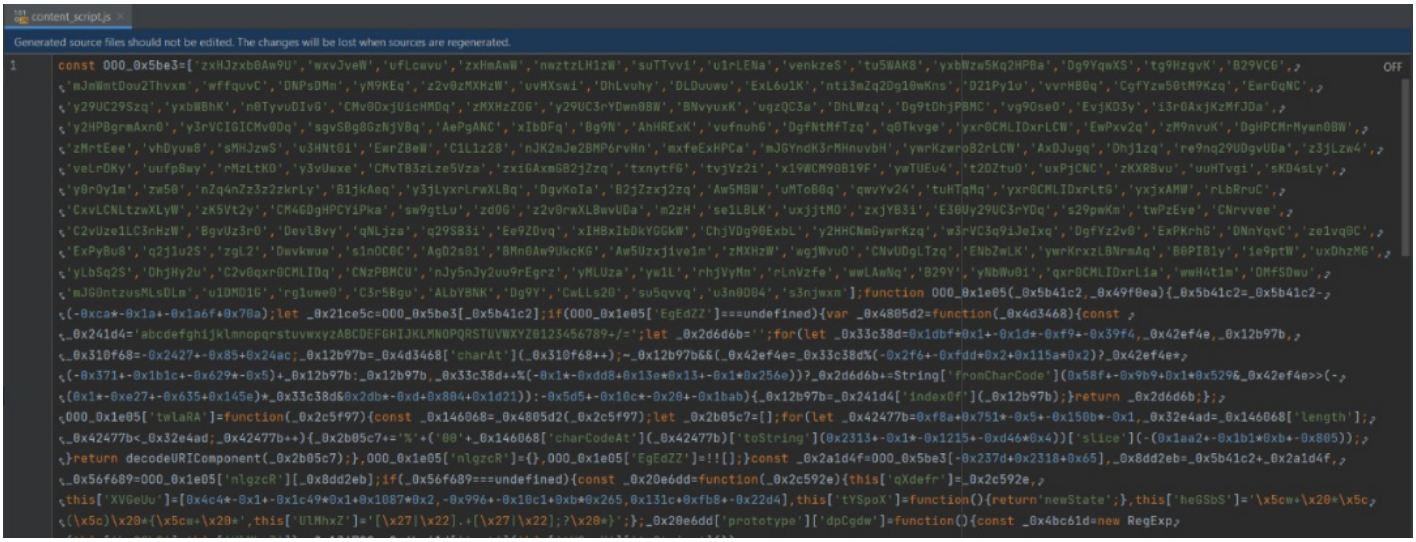
content_script.js 这个脚本会插入到用户访问的每一个页面中并执行

其中，background_script.js比较简单：

```
// Put all the javascript code here, that you want to execute in background.
chrome.runtime.onMessage.addListener(
  function(request, sender, sendResponse) {
    console.log(sender.tab ?
                "from a content script:" + sender.tab.url :
                "from the extension");
    if (request.getflag == "true")
      sendResponse({flag: "OOO{}"});
  }
);
```

增加了一个事件监听器，在收到消息后，如果`request.getflag`为`true`，则调用回调函数`sendResponse`，把flag传进去。大概这样的逻辑，所以猜测content_script.js中应该会有发送消息的函数，然后我需要想办法调用到，这样拿flag。

那么，看下content_script.js……是一个混淆过的JavaScript。基本可以确定这道题的考点就是反混淆了。



0x02 一些失败的尝试

defcon第一天去公司和其他同学一块做，但是一整天都没有Web题，做Misc没啥头绪，Pwn直接不会做，于是打了一天酱油，第二天就没去公司。结果第二天下午在家吃完饭上去看到别人说出来一道Web题目，我就自己在家做了。

自己一个人做题最大的问题就是容易钻牛角尖，我拿到大段混淆的代码想的第一件事不是调试，而是优化一下（其实作用不大）。

代码里面有很多类似`0x2186+-0x1*-0x1523+-0x36a9`这样的纯数字表达式，另外还有一些`'return\x20/\x22\x20'+'+\x20this\x20+\x20\x22'+'/'`这样的字符串表达式，其实都是可以优化成一个简单表达式的。

美化的思路其实比较简单，就是对代码的AST树进行遍历，遇到`BinaryExpression`节点就执行这个表达式，然后用结果生成一个新的`Literal`节点替换掉原来的`BinaryExpression`节点，最后生成新的代码。

以前给X-Ray做XSS扫描器用的比较多的AST Parser是**Esprima**，但是esprima中缺少几个东西：

esprima自带的walker（`parseScript`第三个参数），不支持替换节点

替换后的新AST树，无法还原为代码

所以，还需要我自己解决一下这两个问题。第一个问题，可以编写一个遍历AST树的函数，这里直接用递归做就行了。调用回调后，如果有返回值，则将这个返回值替换掉原本的节点，并停止继续递归；如果没有返回值，则继续递归，代码如下：

```
function walk(ast, fn) {
    for (let i in ast) {
        let child = ast[i];
        if (child && typeof child.type === 'string') {
            let newNode = fn(child);
            if (newNode) {
                ast[i] = newNode;
            } else {
                walk(child, fn);
            }

        } else if (child instanceof Array) {
            for (let j in child) {
                let childchild = child[j];
                let newNode = fn(childchild);
                if (newNode) {
                    child[j] = newNode;
                } else {
                    walk(childchild, fn);
                }
            }
            ast[i] = child;
        }
    }

    return ast;
}
```

然后，我写了一个简单的回调，其作用是将一些代码里奇奇怪怪的表达方式替换成原本的值，比如`\x61\x62\x63\x64`替换成`abcd`，`0x4521`替换成十进制`17697`：

```
ast = walk(ast, node => {
    if (node.type === 'Literal') {
        if (node.value === null) {
            node.raw = 'null'
        } else {
            node.raw = node.value.toString();
        }
        return node;
    }

    return null;
});
```

第二个问题，esprima没有将AST树转换成代码的功能，需要借助另一个库recast。用法比较简单，`recast.print(node)`。

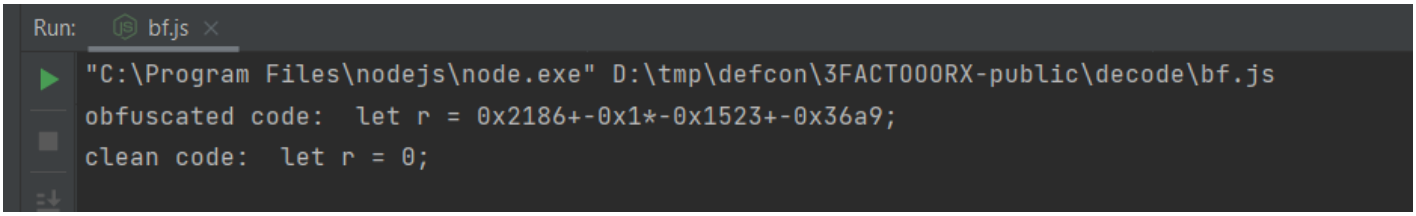
我另一个需求是将纯数字字符串的节点合并，比如`0x2186+-0x1*-0x1523+-0x36a9`计算出来结果0，那么就用0替换这个表达式。

实现方法也很简单，当发现节点类型是`BinaryExpression`，则说明进入了我需要的节点。但是，此时还需要判断一下这个节点的子节点中不能有变量之类的其他对象，否则是无法计算的。我这里写了一个`canEval`函数，用了非递归的形式搜索了叶子节点，如果有非`Literal`的节点，则返回false：

```
function canEval(ast) {
    let queue = [ast];
    while (queue.length > 0) {
        let node = queue.shift();
        if (node.type === 'Literal') {
            // do nothing
        } else if (node.type === 'UnaryExpression') {
            queue.push(node.argument);
        } else if (node.type === 'BinaryExpression') {
            queue.push(node.left);
            queue.push(node.right);
        } else {
            return false;
        }
    }
    return true;
}

ast = helper.walk(ast, node => {
    if (node.type === 'BinaryExpression' && helper.canEval(node)) {
        let result = eval(recast.print(node).code);
        return {
            type: 'Literal',
            value: result,
            raw: result
        }
    }
});
```

`canEval`如果返回true，则说明可以进行计算，用recast将其还原成代码，直接用eval执行（这里是否有潜在的安全问题？）。获取计算结果后，生成一个新的Literal节点，替换到AST树上。

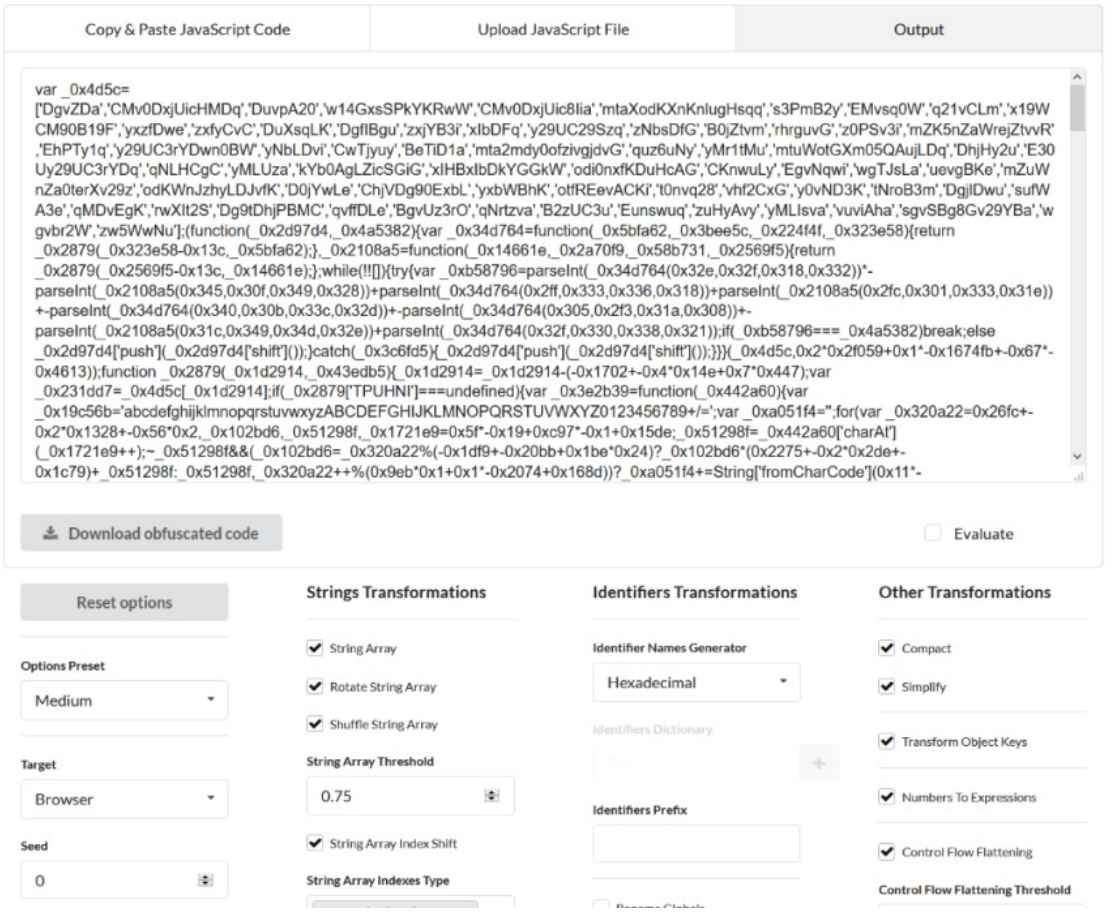
最后实现的效果是：



不过，我做了这一堆东西，实际上对CTF解题帮助不大，最后生成的代码是这样：**https://gist.github.com/phith0n/7e652fb70916166b3b91dc1c14ec934b**，仍然很难看懂。

后面又硬看代码看了一段时间，手工优化了一些函数，个人有点完美主义的感觉，总想把代码读懂。后来偶然刷了一下排行榜，发现已经有好几只队伍做出来了，果断放弃了现在的思路，肯定不对。

换思路，我去网上搜索了一下“javascript obfuscate code”，出来第一个结果**javascript-obfuscator**，这是一个JavaScript混淆工具。我简单试了下，其输出的代码和题目代码非常像，有几个特征点都能对应上：



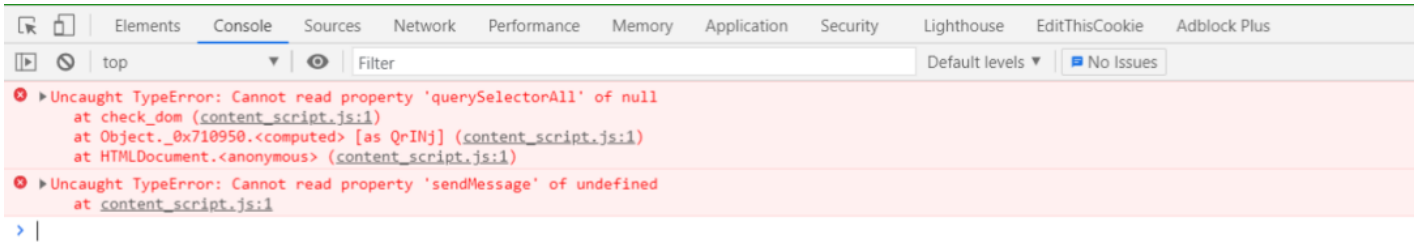
于是我上网上找它的反混淆工具，尝试了几款，都无法正常解密，看起来像是算法被改过了。没有细看具体原因。

这里说一下为什么我一直没有尝试调试，因为之前在阅读优化过的代码时发现，代码里有一些反调试的方法，如果弄不好可能会进入死循环什么的，所以调试的优先级会比较低。但实际上，打CTF无论如何都应该先尝试调试才对，毕竟时间是有限，没空细细研磨代码。

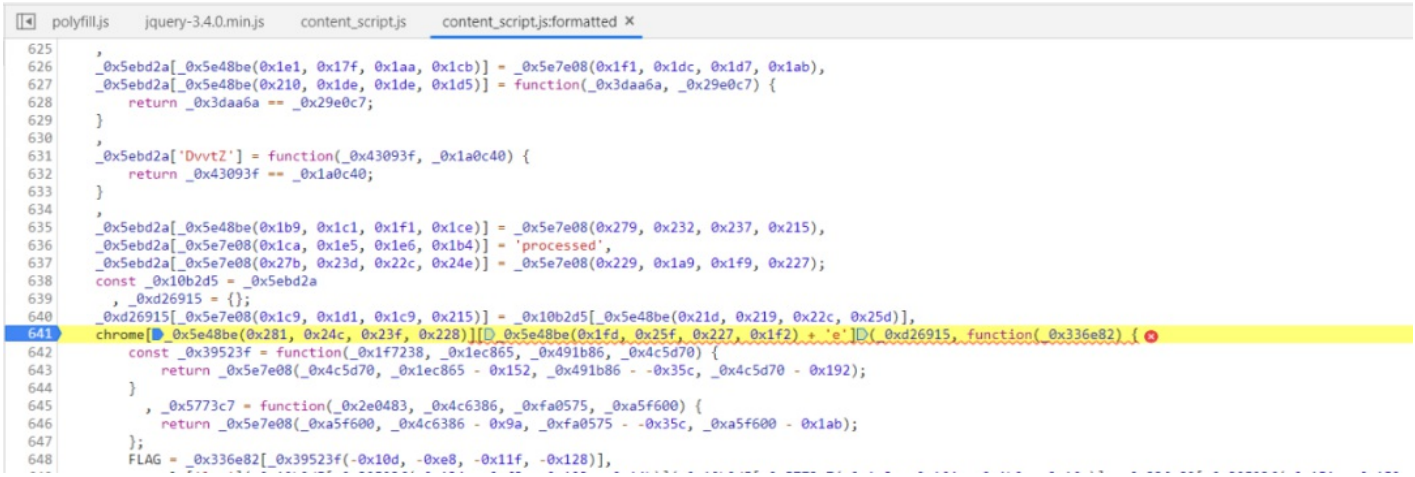
后来实际调试的时候发现并没有触发反调试的机制，这一点也出乎我的意料。

0x03 调试代码

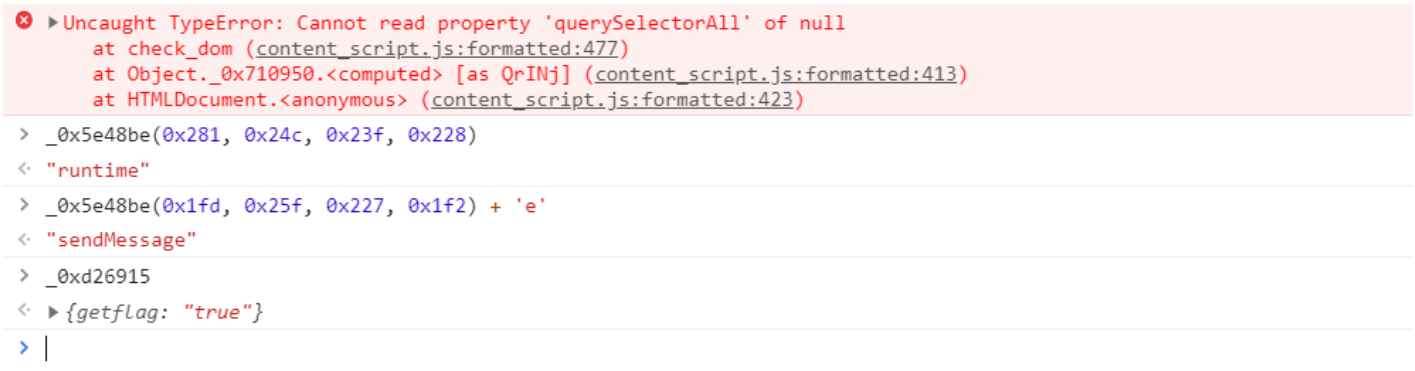
前面一些方法都没解决问题，于是我开始尝试调试代码。一开始调试的时候没有安装浏览器插件，直接将content_script.js作为一个JavaScript加载到一个空白页面里，此时出现了两个错误：



第二个错误引起了我的注意。因为这道题是发送消息给后台监听器获取flag，而这里正好在发送消息的函数`sendMessage`位置出错了。点击控制台，直接定位到出错的代码：



在这里下个断点，刷新页面，断下后就可以计算此时几个函数的值了：

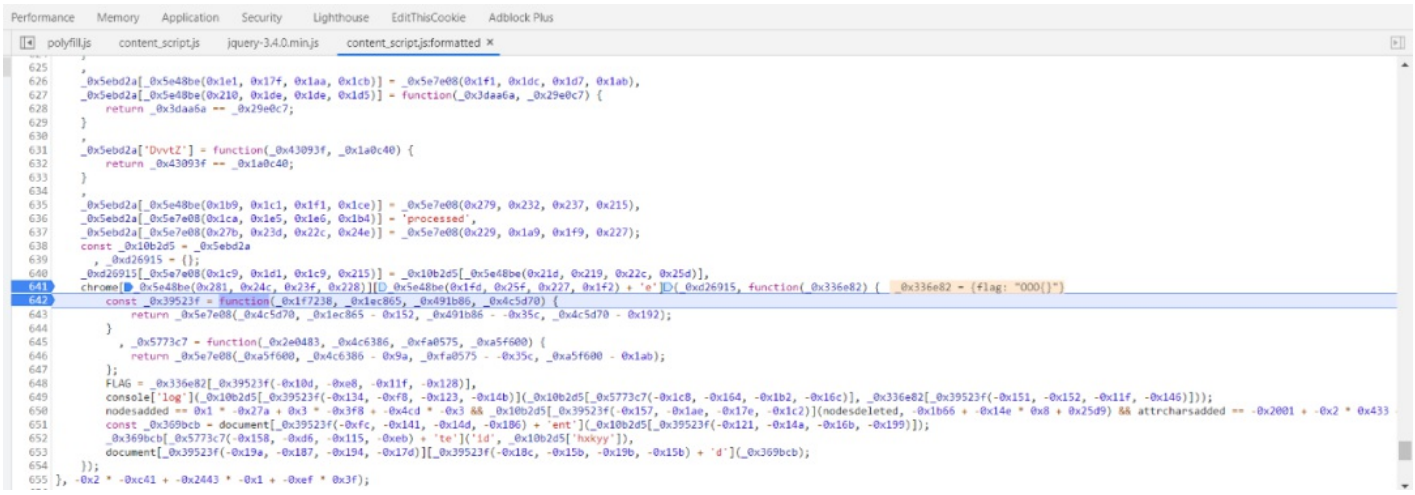


所以，此时这个调用可以简化为：

```
chrome.runtime.sendMessage({getflag: true}, function() {...})
```

很显然，这里出错的原因是，当前脚本没有运行在浏览器扩展的上下文中，所以没有`chrome.runtime`这个对象。于是，我安装了一下浏览器扩展，在扩展页面加载解压后的目录即可。

启用了扩展以后，访问的每一个页面都会执行混淆后的代码。我重新在同一个位置下了个断点，并且在后面的回调里也下了个断点，可见，已经能成功进入回调了：



这说明消息已经成功发送给background_script.js，并且已经成功执行回调。单步调试回调函数中的代码，即可一行一行地进行手工执行，将里面的函数调用替换成返回值。好在这个函数的内容不多，优化后的代码如下：

```
chrome.runtime.sendMessage({getflag: true}, function (_0x336e82) {
    FLAG = _0x336e82['flag'];
    console['log'](_0x10b2d5['KShsG'](_0x10b2d5['bppSB'], _0x336e82['flag']));
    nodesadded == 5 && nodesdeleted==3 && attrcharsadded == 23 && domvalue == 2188 && (document['getElement
    const _0x369bcb = document['createElement']('div');
    _0x369bcb['setAttribute']('id', 'processed'),
        document['body']['appendChild'](_0x369bcb);
})
```

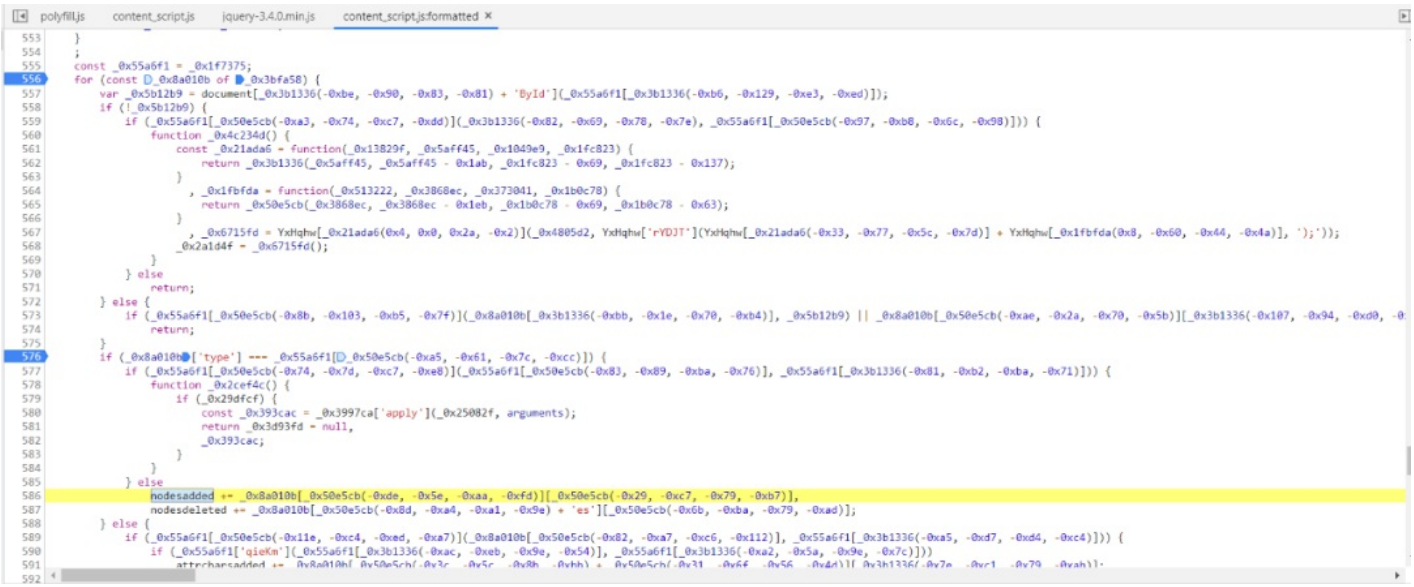
这个比较好懂了，拿到flag以后，且满足这几个条件：`nodesadded == 5 && nodesdeleted==3 && attrcharsadded == 23 && domvalue == 2188`，flag会被写入到id是`thirdfactooor`的DOM节点的value中。

所以，现在要做的就是满足这4个变量的值。

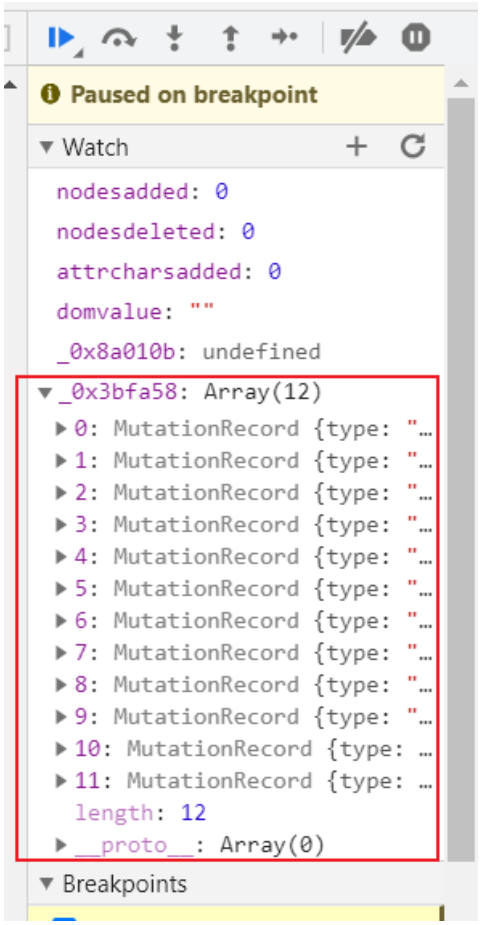
0x04 打怪升级

现在等于有4关，只要依次闯过，就可以获取flag了。

首先，全局搜一下`nodesadded`，找到一处修改的代码。在这行代码前面两个关键的条件位置下个断点：



断在for循环上，此时查看for循环里的变量，是一个由`MutationRecord`对象组成的数组：



搜了一下，这个对象是传给MutationObserver回调的一个参数。MutationObserver我之前写油猴脚本时用过一次，作用是监控某一个DOM节点，看是否有变化，如果有变化，则会触发回调函数。

单步向下执行会发现并不能走到修改nodesadded的地方，单步调试手工分析后，代码简化如下：

```
for (const _0x8a010b of _0x3bfa58) {
    var _0x5b12b9 = document.getElementById('3fa');
    if (!_0x5b12b9) {
        // ...
        return ;
    } else {
        if (_0x8a010b['target'] === _0x5b12b9 || _0x8a010b['target']['parentNode'] === _0x5b12b9 || _0x8a01
            // 需要进入这里
        } else {
            return;
        }
    }

    if (_0x8a010b['type'] === 'childList') {
        nodesadded += _0x8a010b['addedNodes']['length'],
        nodesdeleted += _0x8a010b['removedNodes']['length'];
    }
}
```

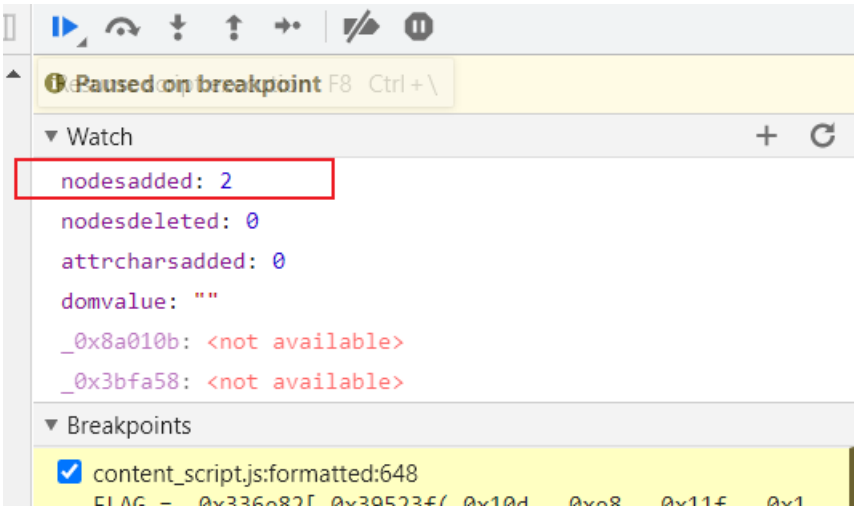
这样就比较好理解了，页面使用MutationObserver监控了DOM变化，如果DOM发生变化，且这个变化的DOM是id为3fa的DOM节点，那么就给增加nodesadded的数字。

简单来说，`#3fa`这个节点下的DOM元素，增加1个，nodesadded就加1；减少1个，nodesdeleted就加1。其实我做题的时候没有分析的这么细，之前看到nodesadded和nodesdeleted这两个名字的时候就猜到和DOM节点的增加删除有关，再结合一点相关的代码就找到了方法。

我写了这样一个HTML页面来测试我的想法：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>test</title>
</head>
<body>
    <div id="3fa"></div>
</body>
<script>
    let a = document.getElementById('3fa');
    a.appendChild(document.createElement('img'));
    a.appendChild(document.createElement('img'));
</script>
</html>
```

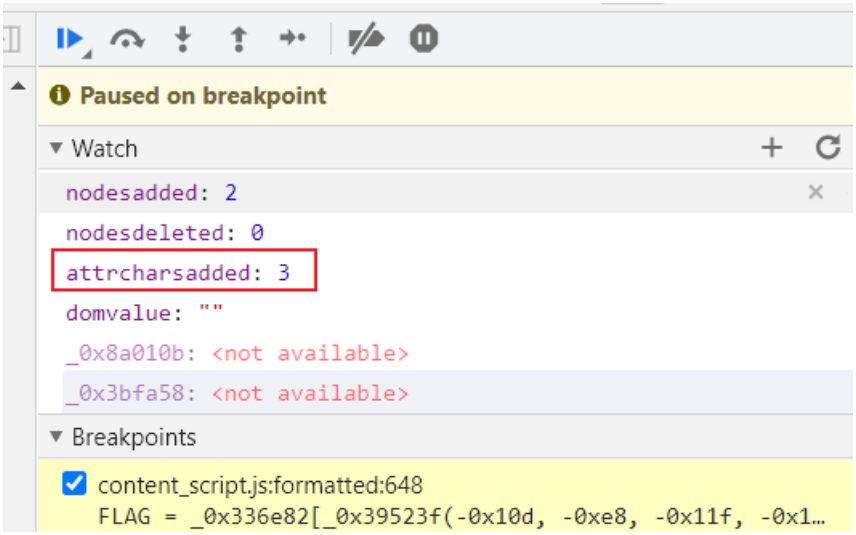
我向`div#3fa`中增加了两个img。此时，在EventListener的回调中下断，可见此时nodesadded就是2，nodesdeleted也类似，这两个变量搞定了：



attrcharsadded，从名字上来看应该和DOM节点的属性的字符数量有关。这次没有分析代码，因为有上次的经验，直接测试了一下修改属性：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>test</title>
</head>
<body>
    <div id="3fa"></div>
</body>
<script>
    let a = document.getElementById('3fa');
    a.appendChild(document.createElement('img'));
    a.appendChild(document.createElement('img'));
    a.setAttribute("abc", '123456');
</script>
</html>
```

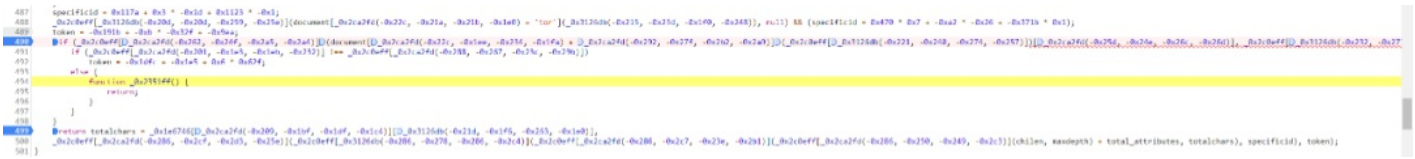
结果是3，看来是abc的长度了，也就是属性的键名：



domvalue，这个变量从名字来看并不能猜出其算法。在代码里搜索了一下，只有一个被赋值的地方：

```
domvalue = _0x4d8085[_0x4bc4df(-0x52, -0x9c, -0x40, -0x65)](check_dom);
// 实际上就是 domvalue = check_dom()
```

domvalue的值来自于check_dom函数的返回值。check_dom函数比较长，但这里有个技巧，我在所有可能导致check_dom函数返回的地方下断点，这样就可以拿到返回值了：



优化了一下从if到return的这几行代码：

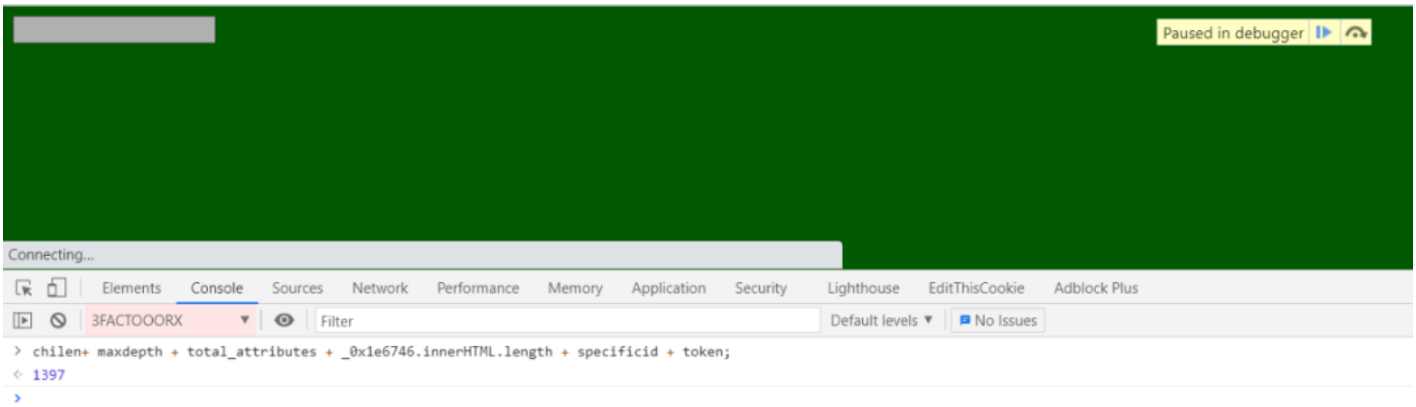
```
var _0x1e6746 = document.getElementById('3fa');
// ...
if (document.querySelector('#thirdfactooor')['tagName'] == 'INPUT') {
    if ('QzIrw' !== 'cunYq')
        token = 1337;
    else {
        function _0x2351ff() {
            return;
        }
    }
}
return chilen+ maxdepth + total_attributes + _0x1e6746.innerHTML.length + specificid + token;
```

前面图中可以看出，if语句出错导致函数退出了，原因是`document.querySelector('#thirdfactooor')['tagName']`这个语句在`#thirdfactooor`不存在时会出现`Cannot read property 'tagName' of null`的错误。所以我构造了一个满足要求的节点：

```
<div id="3fa">
    <INPUT id="thirdfactooor">
</div>
```

此时整个逻辑就清晰了。我其实完全不用关心chilen、maxdepth什么的这些值是怎么算出来的，我要的是`chilen+ maxdepth + total_attributes + _0x1e6746.innerHTML.length + specificid + token`这个整体的值。

在return的位置下个断点，然后计算下当前这个整体的值是多少：



1397。幸运的是，因为这个整体里面有一个值非常好控制，`_0x1e6746.innerHTML.length`，也就是`#3fa`这个div的HTML长度。

后面只需要随便用什么字符填充这个长度到我要的数量就可以了，这个就是domvalue的值。

0x05 构造POC

4个变量的值的来源都弄清楚了，现在就需要构造一个页面，让最后计算出的4个变量满足下面这个条件：

```
nodesadded == 5 && nodesdeleted==3 && attrcharsadded == 23 && domvalue == 2188
```

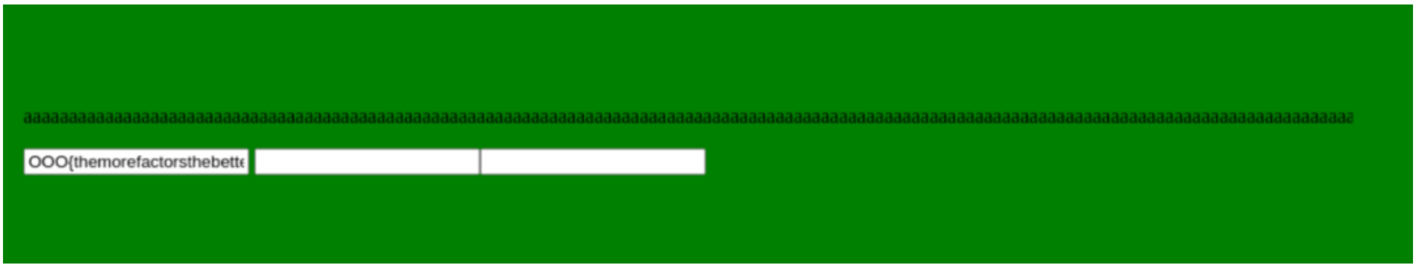
我构造的页面如下：

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width">
  <title>JS Bin</title>
</head>
<body>

<div id="3fa">
  <p>aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
  <INPUT id="thirdfactooor">
</div>

</body>
<script>
  let a = document.getElementById('3fa');
  a.setAttribute('aaaaaaaaaaaaaaaaaaaaaaa', '1');
  for (let i = 0; i < 5; i++) {
    let b = document.createElement('input');
    b.id = 'id-' + i;
    a.appendChild(b);
  }
  for (let i = 0; i < 3; i++) {
    let dom = document.getElementById('id-' + i)
    dom.remove();
  }
</script>
</html>
```

其中：

> 增加了5个input节点，nodesadded等于5
>
> 删除了3个input节点，nodesdeleted等于3
>
> 3fa标签属性键为23个a，attrcharsadded等于23
>
> 用a填充`#3fa`，最后使得domvalue等于2188

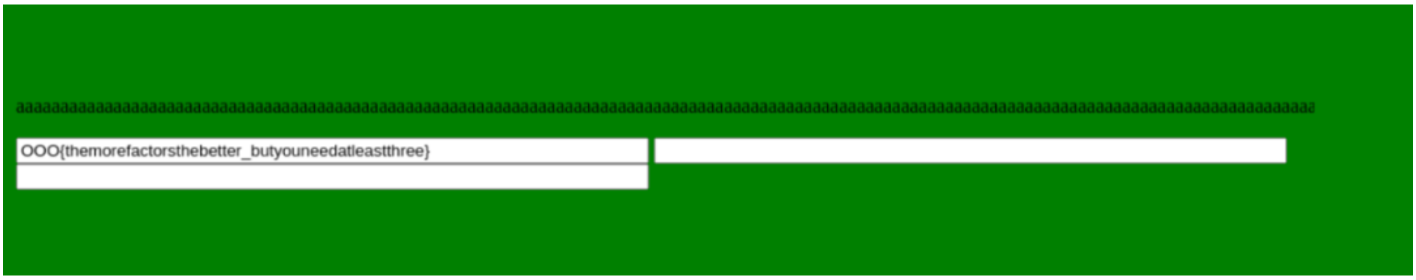
本地测试，可以成功满足条件。上传这个html到题目页面中，结果作为一张图片返回：



可见，已经获取到flag了。但是由于表单长度太短，截图不完整，于是我在CSS里给表单设置了个长度：

```
<style>
    input {
        width: 500px;
    }
</style>
```

再重新尝试，拿到完整flag：



0x06 总结

总结一下，这道题就硬逆，没用到太特殊的方法。教训就是，不要钻牛角尖，如果一条路做了太长时间，就需要休息休息试试其他的路子。

我最后完成题目的时候，看了下已经有十几只队伍提交了flag，尴尬的是我们自己队也已经提交了，应该是几个在公司的Web????做的。这次我纯属打了个酱油，没帮上忙，实在惭愧不已。

本文讲到的JavaScript代码优化的项目，虽然没用上，但是对于以后看代码来说还是有一定帮助的，已经发布在Github：**https://github.com/phith0n/beautifyjs**

谨以此文记录一下。