web逆向攻防-JSVMP自造轮子&正向开发篇

原创 苏心斋|月金剑客 剑客古月的安全屋

作者：yueji0j1anke
首发于公号：剑客古月的安全屋
字数：2025
阅读时间：15min

目录

0x00 前言

前些日子捣鼓了xc（差点被发律师函，已删）和dy以及tx系列的部分jsvmp，比起之前的纯js代码混淆破解难度确实要大，于是琢磨着如何去自制一个jsvmp，完成正向开发。网上基本没有相关资料。。也能理解，毕竟vmp都是各大商业机密，但ast相关还是能自己靠简易教程琢磨清楚的，大概调试开发一周，遂产生了这篇文章

感谢前人指路

https://www.resourch.com/archives/95.html

```
https://github.com/Alanhays/facelessJsvmp
```

0x01 前置知识

1.js执行逻辑

之前我们学xss，讲浏览器的渲染顺序->

```
1.浏览器请求页面，下载html页面、css、js和其他资源
2.解析html，构造dom树
3.遇到script标签，将控制权交给js去控制，修改dom节点，同时还有异步js
```

解码顺序则是

```
1.url解码
2.html解码
3.js解码
```

那对应来说，js的执行逻辑是什么呢

```
1.词法分析
2.语法分析
3.生成ast语法树
4.解析js指令
5.js引擎执行代码
```

这就产生了新的知识，ast语法树

2.ast语法树

ast语法树是编程语言处理一种重要的数据结构，搞过甲方SAST的应该很熟悉了，很多代码审计需要通过解析ast语法树进行相关逻辑漏洞判断，ast类似于一种语法中间结构。

这里提供一个网站


```
https://astexplorer.net/
```

body部分即是相关的代码分析，接下来需要把这些语法转成对应的字节码，那常规来说就是v8去转换解析，随后执行

3.vmp

vmp的流程便是转换成 把这些语法转换成自己对应的字节码

```
1.词法分析
2.语法分析
3.生成ast语法树
4.转换成自生成字节码(虚拟化指令)
5.虚拟机解析
5.js引擎执行代码
```

那我们现在实现自定义vmp需要哪些东西呢

1.自定义指令集

2.源代码转换为字节码

3.解释器

其中，字节码和指令集需要进行一一对应

那当然了，还可以加入代码混淆之内的，这些都是后话了。

准备步骤就绪，让我们开始开发！

0x02 vmp开发

这里我自己写了一套逻辑检验代码

```
Attributes: appName: Netscape;appVersion: 5.0 (Macintosh; Intel Mac OS X
Safari/537.36;platform: MacIntel;userAgent: Mozilla/5.0 (Macintosh; Intel
Safari/537.36;language: zh-CN;webdriver: false;title: 新标签页;URL: chrome
150;devicePixelRatio: 2;Image(): [length: 346, bothExist: true];automatio
false

Hash: -46d2b003

Sign: n7i5m5qq9jltpn2sbr0000000000000000000000000000000000000000000000000
```

存在相关的报错检验、指纹收集、浏览器环境检验、原型链检测、以及相关的模拟器检验

后面的00000都是可以添加的检测项。。。那要将这样一套满是函数、一堆不同语法的js代码去进行vmp，想来还是有点困难。

没关系，一步一步的走。

1.指令集

那首先还是看看指令集，这部分需要进行大量的变量去进行添加，那我们直接设置成列表好了

```
let instructions = []
```

那加入变量，try-catch，if switch function这些都需要进行填入，我这里给个简单示例

```
let instructions = [
    "equal", // 赋值
    "get", // 获取属性
    "def_var", // 声明变量
    "def_Func", // 定义函数
    "params",  // 参数映射
    "return",  // 返回值
    "localScope",  // 当前作用域
    "TExpress",   //console.log相关表达
]
```

当然还有一些运算符。。

你要是涉及到加密算法，包用上的

2.源代码转字节码

需要将ast语法树上的东西进行转换，跟我们的指令集对应上

比如说逻辑表达

```
case "LogicalExpression":
          if (types.isIdentifier(node.left)) {
              sourceToByte(node.left, {constantPool: "lod_var"})
          } else {
              sourceToByte(node.left)
          }
```

```
            sourceToByte(node.right)
            bytecodes.push(IMT[node.operator])
            bytecodes.push(Number(!option.notPushStack))
            break
```

IMT对应是指令映射表，一个指令对应一个字节码

再比如说困了我很久的try catch表达。。我要捕获异常

```
case "TryStatement":
            bytecode.push(IMT['try']);
            t0 = bytecodes.length;
            bytecodes.push(t0)
            bytecodes.length = bytecode.length + 3;
            bytecode[t0 + 1] = bytecode.length;
            sourceToByte(node.block, {specialBlock: 1})
            bytecodes.push(IMT["blockEnd"]);
            bytecodes[t0 + 2] = bytecode.length
            sourceToByte(node.handler.body, {specialBlock: 1})
            bytecodes.push(IMT["blockEnd"]);
            bytecodes[t0 + 3] = bytecode.length
            sourceToByte(node.finalizer, {specialBlock: 1});
            bytecodes.push(IMT["blockEnd"]);
            bytecodes[t0] = bytecode.length
            break
```

一个个对应吧。。。而且对应bytecodes需要跳转，表示指针、开始与结束的位置

3.创造映射

```
let instructionKeysArray = Object.keys(instructionMapTable);
```

```
for (let index = 0; index < instructionKeysArray.length; index++) {
    instructionMapTable[instructionKeysArray[index]] = index;
}
for (let bytecodeIndex = 0; bytecodeIndex < bytecodeArray.length; bytecodeIndex++) {
    let instruction = bytecodeArray[bytecodeIndex];
    if (typeof instruction === "number") continue;
    bytecodeArray[bytecodeIndex] = instructionMapTable[instruction];
}
```

这里一一遍历bytecode，让其变成数字与instruction中的key->value对应，同时instruction中的指令对应的可以随意生成

4.解释器

这部分代码得贴上浏览器的

且作用域可能会混乱，因为涉及到了原型链检测和方法的调用

```
    if(parentScope === 'window') {
        localScope.__proto__ = parentScope
    }
    else {localScope = parentScope;}
```

我这里这样设置。。基本够用了，并且设置对应的栈{}

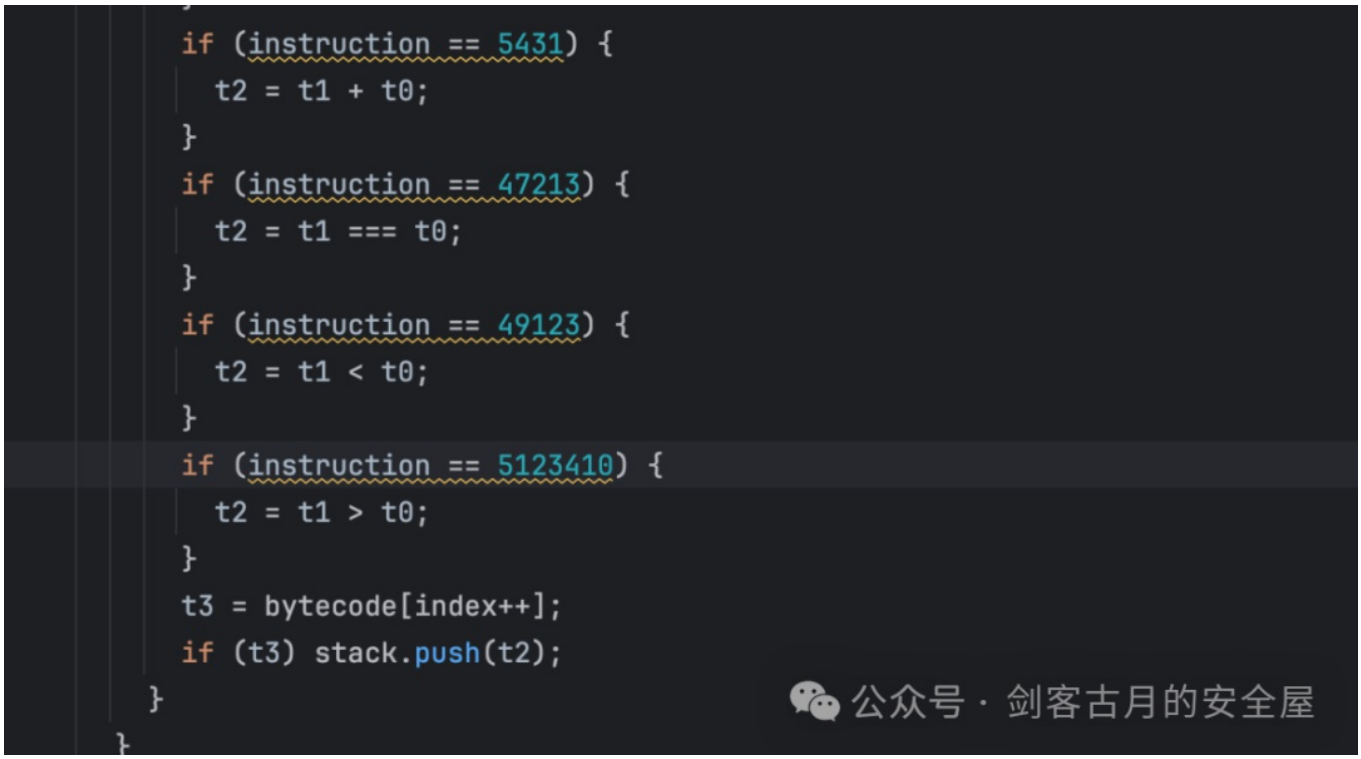
```
parentScope, index, stack, Pool, bytecode, args
```

然后疯狂的switch！！

```
case IMT["<<"]:
case IMT["&"]:......
```

这里对照着手搓了几千行。。。

0x03 效果



这里一看就是运算咳，后期可以增加混淆避免这么直观的发现

0x04 问题

最重要的是。。自身代码的编写

```
    return {
        attributes: attributesList.join(';'),
        hash: simpleHash(combinedAttributes.join(';')),
        sign: sign
    };
}
```


比如我这么编写的时候。。。直观的发现sign发现并没有被成功的赋值，反而把赋值的sign解析成了sign字符串。。还有一堆的属性值无法取的结果。。但这里已经懒得再去手搓代码了，于是直接就赋值或者函数化操作，避免误认。

另外的话就是。符号与操作符一定要补全，不然就不补，不然你都不知道哪里自己给解释错了。。

```
Hash: a834a5f

Sign:
n7i5m5rd3jltpn23br0000000000000000000(
0000000000000000000000000000000000000(
0000000000000000000000000000000000
```


看这效果还不错，用于中小型网站的验签足够了

ob一下瞬间高级了。。