# JavaScript 代码混淆器

金琪琦、刘聪聪

June 12, 2017

# 代码混淆的意义

# 代码混淆的意义

## 一种声音

*“前端代码公开，没有秘密，本身代码就没有保护的意义。”*

# 代码混淆的意义

## 一种声音

*“前端代码公开，没有秘密，本身代码就没有保护的意义。”*

## 我们的观点

- 前端代码天生的不安全性决定了，应该尽可能将重要的业务代码后移动。
- 但一方面，总可能会有一些需要在前端处理，又有一定的敏感性业务；
- 另一方面，前端的一些代码往往是攻击者猜测后端漏洞的入口。

因此对于一些重要的前端代码进行适当的混淆，能够增加攻击者破译的难度。保护前端代码的同时维护整个系统的安全。

# 现状

目前代码混淆在前端使用得并不多。

**原因**

这并不意味着前端代码不需要保护，或者对前端的代码混淆就没有意义。

而是因为前端的大多数代码并不涉及需要高安全的功能，代码混淆必然导致性能损失，对于轻量级的应用性能比安全更重要。

# 应用场景

## 例子

现在越来越多网站的验证码信息不再仅仅通过一张图片，而是从前端采集用户的操作信息返回给后台判断这一系列操作是否属于人类行为。面对这样一个前端代码，一旦知道了它的采集策略就很容易伪造信息。因此对这样重要的前端代码进行混淆是很必要的。

## 淘宝登录代码

淘宝登录界面通过 uab.js 程序来采集用户信息，而这个程序就用来加载一个经过混淆的 JavaScript 程序。

# 设计思路

## 合格的代码混淆器

1. 人力不可识别
2. 增加自动化还原的难度
3. 增加调试的难度

## 实现策略

1. 代码压缩
2. 代码混淆
3. 代码防御

# 程序进度

1. 代码压缩
   - DONE 删除注释
   - DONE 删除空白符
2. 代码混淆
   2.1 变量名替换
   - DONE 全局变量替换为 window 的属性调用
   - DONE 属性调用替换为取元素操作 []
   - DONE 局部变量名随机化

   2.2 常量混淆
   - DONE 提取所有的字符串，通过字符数组打散
   - DONE 常量编码转换

   2.3 控制流替换
   - TODO 将普通的循环语句展开
   - TODO 将顺序执行的代码放置在精心设计的循环之中
3. 代码防御
   - TODO 禁止代码格式化和变量重命名
   - TODO 禁止控制台调试
   - TODO 域名绑定

# 实现简述 1

## 源代码

```
var hello = console.log("hello world");
```

## 全局变量替换为 window 的属性调用

```
this.hello = this.console.log("hello world");
```

## 属性调用替换为取元素操作 []

```
this["hello"] = this["console"]["log"]("hello world");
```

# 实现简述 2

## 提取所有的字符串

```
!function(gin1, gin2, gin3, gin4, gin5) {
  gin1[gin2] = gin1[gin3][gin4](gin5);
}(this, "hello", "console", "log", "hello world");
```

## 字符数组打散

```
!function(Gin) {
  !function(gin1, gin2, gin3, gin4, gin5) {
  gin1[gin2] = gin1[gin3][gin4](gin5);
}(this, Gin(3, 1, 2, 2, 6),
Gin(8, 6, 5, 0, 6, 2, 1),
Gin(2, 6, 7),
Gin(3, 1, 2, 2, 6, 9, 4, 6, 10, 2, 11));
}(function(Gin) {
  return function() {
  for (var t = arguments, r = "", u = 0, i = t.length; i > u; u++)
    r += Gin[t[u]];
  return r;
};
}(["s", "e", "l", "h", "w", "n", "o", "g", "c", " ", "r", "d"]));
```

# 实现简述 3

## 局部变量名随机化

```
!function(I) {
  !function(F, L, H, f, _) {
  F[L] = F[H][f](_);
}(this, I(10, 8, 6, 6, 9), I(0, 9, 3, 4, 9, 6, 8),
I(6, 9, 5), I(10, 8, 6, 6, 9, 7, 1, 9, 11, 6, 2));
}(function(V) {
  return function() {
  for (var B = arguments, c = "", r = 0, $ = B.length; $ > r; r++)
    c += V[B[r]];
  return c;
};
}(["c", "w", "d", "n", "s", "g", "l", " ", "e", "o", "h", "r"]));
```

# 实现简述 4

## 常量编码转换

```
!function(l) {
  !function(c, B, r, n, D) {
  c[B] = c[r][n](D);
}(this, l(0x8, 0x2, 0x6, 0x6, 0x5), l(0x1, 0x5, 0x0, 0x3, 0x5, 0x6, 0x2),
 l(0x6, 0x5, 0x7), l(0x8, 0x2, 0x6, 0x6, 0x5, 0x9, 0xa, 0x5, 0xb, 0x6, 0x4));
}(function(v) {
  return function() {
  for (var q = arguments, t = "", P = 0x0, $ = q.length; $ > P; P++)
    t += v[q[P]];
  return t;
};
}(["\u006e", "\u0063", "\u0065", "\u0073", "\u0064", "\u006f",
"\u006c", "\u0067", "\u0068", "\u0020", "\u0077", "\u0072"]));
```

# 实现简述 5

## 代码压缩

```
!function(U){!function(z,m,f,g,A){z[m]=z[f][g](A);}(this,U(0x3,0xb,0x7,0x7,0x9),U(0x5,0x9,0xa,0x6
,0x9,0x7,0xb),U(0x7,0x9,0x4),U(0x3,0xb,0x7,0x7,0x9,0x1,0x0,0x9,0x2,0x7,0x8));}(function(k){return
 function(){for(var L=arguments,I="",J=0x0,C=L.length;C>J;J++)I+=k[L[J]];return I;};}(["\u0077","
\u0020","\u0072","\u0068","\u0067","\u0063","\u0073","\u006c","\u0064","\u006f","\u006e","\u0065"
]));
```

# 遇见的问题

混淆的核心是在源代码的外部套一个 立即执行函数 。如果源程序中有全局函数的声明，套上立即执行函数后失去了全局性。因此我们将所有的全局 函数声明 转换为 函数表达式 。

```
function hello() {
// do someting
}

var hello = function() {
// do someting
}

!function(a) {
a.hello = function() {
// do someting
}
}(this)
```

但是，函数声明是没有顺序性的，函数表达式必须是顺序的，因此需要根据函数调用的先后顺序来调整函数表达式的顺序。

# 下一步完成

1. 控制流替换
2. 代码防御
3. 主流框架混淆测试

# 控制流替换

## 循环展开

## 将顺序执行的代码放置在精心设计的循环之中

通过 while 和 switch...case 将顺序执行的代码包裹在循环当中

## 难点

暂时不确定需要把哪些部分进行控制流转换

# 将顺序执行的代码放置在精心设计的循环之中

## 顺序执行

```
var sum = 1 + 2;
console.log(1);
console.log(2);
```

# 将顺序执行的代码放置在精心设计的循环之中

## 构造循环

```
var _0x2b972d = {
                '\x42\x6c\x67': function _0x160d18(_0xdc9f31, _0x3741dd) {
                    return _0xdc9f31 + _0x3741dd;
                }
            };
var _0x170490 = '\x35\x7c\x34\x7c\x33\x7c\x36'['\x73\x70\x6c\x69\x74']('\x7c'), _0x4f3437 = 0x0;
while (!![]) {
                switch (_0x170490[_0x4f3437++]) {
                case '\x33':
                    console['\x6c\x6f\x67'](0x2);
                    continue;
                case '\x34':
                    console['\x6c\x6f\x67'](0x1);
                    continue;
                case '\x35':
                    var _0x476f51 = _0x2b972d['\x42\x6c\x67'](0x1, 0x2);
                    continue;
                case '\x36':
                    console['\x6c\x6f\x67'](0x5);
                    continue;
                }
                break;
            }
```

# 禁止控制台调试

在源程序中隐藏插入一段代码，它负责监控程序运行过程中是否启用 console，如果有则抛异常退出。

## 示例代码

```
(function() {
    try {
        var $_console$$ = console;
        Object.defineProperty(window, "console", {
            get: function() {
                if ($_console$$._commandLineAPI)
                    throw " 抱歉，为了用户安全，本网站已禁用 console 脚本功能";
                return $_console$$
            },
            set: function($val$_$) {
                $_console$$ = $val$_$
            }
        })
    } catch ($ignore$$) {
    }
})();
```

# 域名绑定

在源程序中隐藏插入一段代码，它负责监控程序运行时，当前域名是否与既定域名一致，不一致则抛出异常退出。

## 示例代码

```
(function() {
        var chars = '119119119461161011151164699111109';
// 这段就是 www.test.com, 将每个字符转换成了 Unicode 编码
        var hosts = location.host;
        var s = '';
        for(var i=0;i<hosts.length;i++) {
            s += hosts[i].charCodeAt(0);
        }
        if(s != chars) {
            throw new URIError(' 随便写点啥哇。。');
        }
    })();
```

# 禁止代码被修改

在源程序中隐藏插入一段代码，它负责对混淆后的代码进行哈希验证，如果与既定的哈希码不一致，则抛出异常退出。