基础1–JS逆向基础

1 定位 hook 跟值混淆

2 part1 定位

本处的定位不是找函数入口，而是教大家怎么不走冤枉路。这是我们对抗最直接的要求，不要拿到自己公司的网站，上来就找函数入口开始分析。做事一定要有章法和逻辑。

接下来我会简单的描述一下我做这个工作的一般流程（不必遵循，不必循规蹈矩，看自己的理解）。

难度	步骤
最理想状况	1. 重放攻击 ---- 有效 2. 删参数重放攻击 ---- 寻找最优解（最低暴露值，更便于隐藏身份）【求稳，参数全放也无可厚非】 3. 稳定性测试（几小时/几天）
比较理想的情况	1. 重放攻击 ---- 有效 2. 删参数重放攻击 ---- 寻找最优解（最低暴露值，更便于隐藏身份）【求稳，参数全放也无可厚非】 3. 关键参数处理（找接口/固定值/set-cookie【一定要优先！优先！优先！】/js逆向） 4. 流程完善与代码总结 5. 稳定性测试（几小时/几天）

一般糟心的情况	1. 重放攻击 ----- 无效 2a. 关键参数猜测与处理（找接口/固定值/set-cookie【一定要优先！优先！优先！】/js 逆向） 2b. 测试请求 ..... 3. 删参数 ---- 寻找最优解（最低暴露值，更便于隐藏身份）【求稳，参数全放也无可厚非】 4. 流程完善与代码总结 5. 稳定性测试（几小时/几天）
比较糟心的情况	1. 重放攻击 ----- 有效 2. 关键参数处理（找接口/固定值/set-cookie【一定要优先！优先！优先！】/js 逆向） 3. 删参数重放攻击 ---- 寻找最优解（最低暴露值，更便于隐藏身份）【求稳，参数全放也无可厚非】 4. 流程完善与代码总结 5. 稳定性测试（几小时/几天） 6. 短期时间内不明原因崩溃 7a. 重新查看是否为参数问题 7b.填上参数稳定性测试（几小时/几天） .....
非常糟心的情况	1. 重放攻击 ----- 无效 2a. 关键参数猜测与处理（找接口/固定值/set-cookie【一定要优先！优先！优先！】/js 逆向） 2b. 测试请求 ..... 3. 删参数 ---- 寻找最优解（最低暴露值，更便于隐藏身份）【求稳，参数全放也无可厚非】 4. 流程完善与代码总结 5. 稳定性测试（几小时/几天） 6. 短期时间内不明原因崩溃 7a. 重新查看是否为参数问题 7b. 填上参数稳定性测试（几小时/几天） .....

📌 不要一根筋，要知道所有的一切都是选择，不同的选择带来的是不同的风险和收益。【没有绝对意义的最优解，只有相对最优】

3 part2 hook 姿势

3.1 函数部分

a. 简单函数 hook（非对象属性调用，不涉及 this 指向）

eval hook 上节课讲了不再浪费时间

b. 对象属性调用函数（涉及 this 指向）

寻找对象的原型，从原型角度 hook 上节课讲了，但是我还是要浪费时间再讲

例：hook 举例

hook: document.createElement

document.appendChild

全数组 push 方法改写

全函数定义后，自动获得：name 方法【相当于给空函数设置方法】

3.2 Q&A

浏览器上所有的函数都能 hook/重写么？极大多数可以，极个别函数可能会无法重写

hook 函数能被检测到么？ 当然可以了 -----> .toString() 即可

那么：重写 toString 方法就可以了么？那就让你知道这个世界的恶意

(function(){}).toString.call(eval)

那么如何对抗呢?

4 对象部分

4.1 a.对象属性：

一般采用修改其描述符的方法进行 hook

最常见的 hook 方案： hook cookie

由于 cookie 是 document 里面的属性，所以我们可以修改 document 这个对象中 cookie 属性的描述符

网上的例子已经非常成熟了。只讲几个 hook 时的注意事项。这个实现就自己去看吧

https://zhuanlan.zhihu.com/p/231651573

注意事项： 小心递归栈溢出（所以必须有中间变量/缓存接受值）

document.cookie 不是简单的赋值。而是对值的特殊拼接形式

4.2 Q&A

描述符的修改会被检测么?

当然会：Object.getOwnPropertyDescriptor

如何对抗：hook 描述符检测函数啊

那能检测描述符检测函数是否被 hook 么?

它是函数，toString 当然能检测了。

.....

4.3 b. 对象本体的 hook

一般情况下会选择使用 ES6 的新语法： 代理 Proxy
PS：代理是一种非常完善的机制。它几乎无法被检测（不敢说死，但是还没有遇到过检测方案）

```
纯文本
global = new Proxy(window, {
  get: function(){
    console.log(arguments)
  }
})
```

文档在这里： https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy

4.4 Q&A

对象都可以 hook 么?

一部分可以，剩下的不可以。尤其是与 window 相关的内容

比如 location【控制网页 url 的】

document【控制文档流的】

window【控制全局 js 的】

hook 以后我们边实战边继续带着大家回忆。它大概率会一直伴随着我们了。

5 part3 常规跟值基础

5.?.1 同步

【默认已经找到了任意一处有密文的位置】

同步跟值其实没什么太多的技巧。核心原则就是找来源。现在断点已经看到密文了

我们都知道堆栈代表着什么。所以，只要在断点处看到了要跟的值

向上看，上面也没有定义，是 arguments 传进来的

那就一定向上找堆栈呗 ~

然后 again，again，again。一直找到来源就可以了【这个过程可能也会非常繁琐哦】

5.?.2 异步

【默认已经找到了任意一处有密文的位置】

异步跟值跟同步跟值差不多，但是区别是异步是非常容易跟丢的。就是指，某个指跳到上一个堆栈突然就消失了并且上一层堆栈的函数传值里也没有这个玩意，就好像这个是凭空产生的一样

这个时候我一般常用的一个技巧：

就在这个凭空产生的位置下断点，然后 Step 调试进异步，看微队列里执行（这个过程会非常复杂和繁琐，跟的时候手会疼）

5.?.3 毒瘤:jsvmp 虚拟化保护技术

特殊跟值技巧：

实战中逐渐给大家讲。这个确实有些偏经验向。比较吃基础水平。这里给大家举几个例子大家就明白了

特殊：代表着不具有泛用性，只能是针对某些情况的处理。遇不到我其实也很难总结出来。

比如：

a. 看到了一个含有密文的内容 a,是一个构造函数 A 的实例化对象

b. 看到了一个数组 [1937774191, 1226093241, 388252375, 3666478592, 2842636476, 372324522, 3817729613, 2969243214]

c. 看到了一个函数名为：hex_md5

d. 看到了这样一段代码：o'toString';

d0['RC4']['decrypt'](y, d1, {});

d0['enc']['Utf8']'parse';

e. 看到了 axios 特征

f. 看到了 $.ajax

g. 看到了 (!function(){*******})("很长一段")

h. 看到了大数组 + 数组移位 + 解密函数

PS：这些所谓的"经验"才是真正区分逆向高手和逆向菜逼的标尺

既然是标尺、也就代表着一定会分流。有人能、有人不能、有人学的慢、悟不出来。有人学的快、

悟得快

同样：我也不能保证我的做法是最优解。但是我保证会把我知道的东西和情况，都在之后的实战课中告诉大家

📌 Q&A 为什么先讲跟值，而不去完整的讲加密函数入口搜寻。（因为函数入口是个大活儿。不是几条规则总结起来就能说明白的）

所以我们实战中讲。遇到一个讲一个。

6 part4 混淆的基础与目的

混淆主要的目的就是增加代码的阅读难度和调试难度，增加对抗的强度，混淆的主要方法就是把一段代码变得难以理解，用肉眼难以看得出来。

常见的混淆手段有：

6.1 变量名与变量不可视

```
青铜: a,b,c,d
```

```
黄金: _0x5c97ce = [0x27a]
```

```
王者: eval(String.fromCharCode('1766') + '=100')
```

6.2 控制流平坦化

```
a = 10;
b = 50;
c = 100;
d = a+b+c;
平坦处理后:
_ip = 0;
while (1) {
  if (_ip === 1) {
    b = 50;
    _ip++
  };
  if (_ip === 0) {
    a = 10;
    _ip++
  };
  if (_ip === 3) {
    d = a + b + c;
    break;
  };
  if (_ip === 2) {
    c = 100;
    _ip++
  };
}
```

6.3 花指令

js 花指令通常指的是增加一些无意义的低耗能运算在逻辑中，增加调试复杂度（这个和真正意义的上的花指令有一定区别，我们不用管）

还是刚刚那个例子。它可以增加其他的执行逻辑在里面

```
纯文本
a = 10;
b = 50;
g = function(){
  a = 10;
  b = 50;
  c = 100;
  d = a+b+c;
}
c = 100;
d = a+b+c;
delete d
```

6.4 死代码

指不运行的代码。还是刚才那个逻辑，它加入了 g 函数，但是这个函数从出生开始就没有用过,这样的话，我们随机断点的时候，就很有可能打到无用的代码导致断不住。

7

死代码注入高手可以将代码和加密的内容的名字写得完全一致。导致你可以搜出上千，上万个关键词

```
纯文本
a = 10;
b = 50;
g = function(){
    a = 10;
    b = 50;
    c = 100;
    d = a+b+c;
}
c = 100;
d = a+b+c;
delete d
```

6.5 三目表达式干扰

```
纯文本
(function (f) {
for (var r = 1; ;) if (r < 4) if (r < 2) r < 1 ? (c = 100, r += 4) : (function (f)
    for (; ;) return
}, r += 4); else {
    if (!(r < 3)) return;
    b = 50, r -= 2
} else r < 5 ? (d = a + b + c, r -= 1) : (a = 10, r -= 3)
})();
```

6.6 代码压缩成一行 + 格式化的检测

```
纯文本
```

```
( toString )
```

等等.....

上面的代码，虽然它干扰调试，但是上下文执行顺序是没有被干扰的。也就是说，虽然是一直跳来跳去，但是代码

还是按照最开始混淆之前的那个样子按照顺序正常执行下来的。但是 jsvmp 可就不太一样了。

jsvmp：全称 Virtual Machine based code Protection for JavaScript，即 JS 代码虚拟化保护方案。

八股文很多，但是了解很必要！ 如果要是不理解一些运行基础的话，它理解起来会比较吃力。所以这节课，我们浅浅的给大家介绍一下

目的就是让大家心里先有个数，jsvmp 它到底在干些什么。以后我们再说处理它的方法。

首先，常见的 jsvmp 的实现方法，你可以理解为：就是自己写了一段代码解释器，用来解释自己的代码

而这个自己的代码：可以是密文，也可以是所谓的明文

我们先不用管所谓的 寄存器 指针 指令 是什么东西。先给大家分析一段简单的代码

```
纯文本
var a = 1000;
var b = 1000;
var c = 1000;
var d = a + b;
var e = d + c;
```

就这段代码，我们现在去理解它做了什么;

用我们人类的语言对它进行翻译(我们先不考虑变量提升那些乱七八糟的事儿):

```
纯文本
声明一个变量 a, 并且给a赋值 1000;
声明一个变量 b, 并且给b赋值 1000;
声明一个变量 c, 并且给c赋值 1000;
声明一个变量 d, 计算 a+b 的值, 将值赋值给 d;
声明一个变量 e, 计算 d+c 的值, 将值赋值给 d;
```

那么下面，我们在把下面的分段再翻译成代码:

```
纯文本
var a;
a = 1000;
var b;
b = 1000;
var c;
c = 1000;
var d;
? = a + b;
d = ?;
var e;
e = d + c;
```

OK, 我们完成了， 下面我们开始定义指令

我们假设赋值指令为 66, 加和指令为 88,声明指令为 110(爱取什么名字取什么名字)

如果按照从上到下的顺序，我们就可以将他们的操作变成指令性的|用 | 分割左侧和右侧」

```
110 --- var | a
66 --- a |  1000
110 --- var | b
66 --- b |  1000
110 --- var | c
66 --- c |  1000
110 --- var | d
88 --- a | b  -----> ?
66 --- d | ?
110 --- var | e
88 --- d | c   -----> ?
66 --- e | ?(此处的d 与 c的和)
```

下一步,我们将每一步，都压入一个数组中，我们就得到了这样一个数组：

```
_stack = [
    [110, 'var', 'a'],
    [66, 'a', 1000],
    [110, 'var', 'b'],
    [66, 'b', 1000],
    [110, 'var', 'c'],
    [66, 'c', 1000],
    [110, 'var', 'd'],
    [88, 'a', 'b'],
    [66, 'd', '?'],
    [110, 'var', 'e'],
    [88, 'd', 'c'],
    [66, 'e', '?'],
]
```

然后我们就可以靠接下来这个数组去完成我们刚才的代码了，其实这个代码写起来就相当简单了

```
var register;    // 这个就当做是问号的存储位置
var variable = {};    // 这个就当做是var变量的存储位置。由于没有其他声明方式的存在，所就不写其他的了
for(let i=0;i<_stack.length;i++){

    instruct = _stack[i][0]   // 我这么写是让大家看得懂，真正的时候，人家才不会那么好心
    left = _stack[i][1]   // 我这么写是让大家看得懂，真正的时候，人家才不会那么好心
    right = _stack[i][2]   // 我这么写是让大家看得懂，真正的时候，人家才不会那么好心

    if(instruct === 110){
        variable[right] = '';
    }
    if(instruct === 66){
        if(right === '?'){
            variable[left] = register;
        }
        else{
            variable[left] = right;
```

```
        }
    }
    if(instruct === 88){
        register = variable[left] + variable[right];
    }
}
var variable = {};    // 这个就当做是var变量的存储位置。由于没有其他声明方式的存在，所就不写其他的了
```

这样代码就写完了，然后我们在简单的做一些加工，封装到自执行函数里,注释去掉,再压缩一下

```
纯文本
!function(_stack){var register;var variable={};for(let i=0;i<_stack.length;i++){instruc
'a''charCodeAt'
```

这还是基于我只写了三个操作

那其他的呢。所有的操作都压成这个样子呢?

我在把传进来的数组按照某种逻辑随机呢?

我把数组分类开来呢?

可操作空间是不是太多啦