SpiderMonkey 漏洞利用简介（一）

f0**** / 2019-08-26 08:45:00 / 浏览数 3194 安全技术 二进制安全 顶(0) 踩(0)

注：本文是一篇翻译文章，原文链接：https://doar-e.github.io/blog/2018/11/19/introduction-to-spidermonkey-exploitation/#jsvalues-and-jsobjects
由于文章比较长，译者将其分为三部分，第一步分解释基础的知识点和背景；第二部讲解利用过程，第三部分结论及其他内容。

介绍

这个博客文章介绍了针对 SpiderMonkey JavaScript Shell 解释器和 Windows 10 RS5 64 位 Mozilla Firefox
的三个漏洞的开发，从一个从未编写过浏览器漏洞利用的人的角度来写，也没有密切关注任何 JavaScript 引擎代码库。
您可能已经注意到，在过去的一两年里，人们对利用浏览器产生了很大的兴趣。每个主要的 CTF
竞赛至少有一次浏览器挑战，每个月至少有一次或两次涉及浏览器利用。这就是为什么我认为我应该从内部分析一下 JavaScript
引擎的内容，然后写一个其中的利用。我选择了 Firefox 的 SpiderMonkey JavaScript 引擎和由 itszn13 出的挑战题 Blazefox。
在这篇文章中，我介绍了我的发现和我在此挑战中编写的三个利用。最初，挑战是针对 Linux x64 环境，但我决定在 Windows x64
上利用它。现在你知道为什么有 3
种不同的利用了吗？三个不同的利用允许我一步一步地实现，而不是同时面对所有复杂的情况。这通常是我日常工作的方式，我只是做了一些小工作，慢慢迭代然后建立起来
以下是我如何组织这些事情的方法：

- 第一件事，我写了一个名为 sm.js 的 WinDbg JavaScript 扩展，它让我可以看到 SpiderMonkey
  中的一堆东西。熟悉对象在内存中组织的各种方式也是一种很好的练习。这不是必要操作，但在编写漏洞时它一定非常有用。
- 第一个利用部分：basic.js，针对 JavaScript 解释器 js.exe
  的一个非常具体的构建。它里面写满了硬编码的偏移，并且没有办法到我的系统以外的其他地方使用这个特定版本的 js.exe 。
- 第二个利用部分，kaizen.js，是对 basic.js 的改进版。 它仍然以 JavaScript 解释器本身为目标，但这一次，它能动态地解决了一大堆东西。 它还使用
  baseline JIT 让它生成 ROP gadgets。
- 第三个利用部分，ifrit.js，最后用一点额外的操作定位Firefox浏览器。我们就不仅是利用 baseline 来生成一个或两个 ROP gadgets，而是将 JIT
  编写到一个完整的 payload 中。不需要 ROP 链，直接扫描以查找 Windows API 地址或创建可写和可执行的内存区域。我们只需要将执行的 payload 放到 JIT
  代码的位置。对于了解 SpiderMonkey 并且已经写过浏览器利用的人来说，这可能是一个不那么枯燥，且有趣的部分。

在开始之前，对于那些不想阅读整个文章的人：我已经创建了一个blazefox GitHub存储库，您可以克隆所有材料。 在存储库中，还可以找到：

- sm.js 这是上面提到的调试器扩展
- 3个利用部分的源码
- JavaScript shell 的64位调试版本，js-asserts 中的私有符号信息，以及 js-release 中的发布版本
- 用来在脚本中构建你自己 payload 技术的脚本
- 用于构建 js-release 的源代码，以便您可以在 src/js 中的 WinDbg 中进行源代码级调试
- 一个 64 位版本的 Firefox 二进制文件以及 ff-bin.7z.001 和 ff-bin.7z.002 中 xul.dll 符号信息
  好，让我们开始吧。

Setting it up 设置它

当然，必须设置一个调试环境。 我建议为此创建一个虚拟机，因为您将不得不安装一些您可能不想在您的个人计算机上安装的东西。
首先，我们先拿到源码。Mozilla 使用 mercurial 进行开发，但它们也维护一个只读的 GIT
镜像。我建议只是克隆这个存储库，这样更快（存储库大约~420MB）：

```
>git clone --depth 1 https://github.com/mozilla/gecko-dev.git
Cloning into 'gecko-dev'...
remote: Enumerating objects: 264314, done.
remote: Counting objects: 100% (264314/264314), done.
remote: Compressing objects: 100% (211568/211568), done.
remote: Total 264314 (delta 79982), reused 140844 (delta 44268), pack-reused 0 receiving objects: 100% (264314/26431
Receiving objects: 100% (264314/264314), 418.27 MiB | 981.00 KiB/s, done.
Resolving deltas: 100% (79982/79982), done.
Checking out files: 100% (261054/261054), done.
```

现在，我们只对创建 JavaScript Shell 的解释器感兴趣，这只是 SpiderMonkey 树的一部分。js.exe 是一个可以运行 JavaScript 代码的简单命令行程序。
编译速度要快得多，但更重要的是它更容易攻击和推理。我们已经准备好放弃代码，所以让我们首先关注一些细微的东西。
在编译之前，抓取 blaze.patch 文件（暂时不需要理解）：

```
diff -r ee6283795f41 js/src/builtin/Array.cpp
--- a/js/src/builtin/Array.cpp  Sat Apr 07 00:55:15 2018 +0300
+++ b/js/src/builtin/Array.cpp  Sun Apr 08 00:01:23 2018 +0000
@@ -192,6 +192,20 @@
    return ToLength(cx, value, lengthp);
}
```

```
+static MOZ_ALWAYS_INLINE bool
+BlazeSetLengthProperty(JSContext* cx, HandleObject obj, uint64_t length)
+{
+    if (obj->is<ArrayObject>()) {
+        obj->as<ArrayObject>().setLengthInt32(length);
+        obj->as<ArrayObject>().setCapacityInt32(length);
+        obj->as<ArrayObject>().setInitializedLengthInt32(length);
+        return true;
+    }
+    return false;
+}
+
+
+
/*
 * Determine if the id represents an array index.
 *
@@ -1578,6 +1592,23 @@
    return DenseElementResult::Success;
}

+bool js::array_blaze(JSContext* cx, unsigned argc, Value* vp)
+{
+    CallArgs args = CallArgsFromVp(argc, vp);
+    RootedObject obj(cx, ToObject(cx, args.thisv()));
+    if (!obj)
+        return false;
+
+    if (!BlazeSetLengthProperty(cx, obj, 420))
+        return false;
+
+    //uint64_t l = obj.as<ArrayObject>().setLength(cx, 420);
+
+    args.rval().setObject(*obj);
+    return true;
+}
+
+
// ES2017 draft rev 1b0184bc17fc09a8ddcf4aeec9b6d9fcac4eafce
// 22.1.3.21 Array.prototype.reverse ( )
bool
@@ -3511,6 +3542,8 @@
    JS_FN("unshift",            array_unshift,      1,0),
    JS_FNINFO("splice",         array_splice,       &array_splice_info, 2,0),

+    JS_FN("blaze",            array_blaze,      0,0),
+
    /* Pythonic sequence methods. */
    JS_SELF_HOSTED_FN("concat",      "ArrayConcat",      1,0),
    JS_INLINABLE_FN("slice",    array_slice,        2,0, ArraySlice),
diff -r ee6283795f41 js/src/builtin/Array.h
--- a/js/src/builtin/Array.h    Sat Apr 07 00:55:15 2018 +0300
+++ b/js/src/builtin/Array.h    Sun Apr 08 00:01:23 2018 +0000
@@ -166,6 +166,9 @@
array_reverse(JSContext* cx, unsigned argc, js::Value* vp);

extern bool
+array_blaze(JSContext* cx, unsigned argc, js::Value* vp);
+
+extern bool
array_splice(JSContext* cx, unsigned argc, js::Value* vp);

extern const JSJitInfo array_splice_info;
diff -r ee6283795f41 js/src/vm/ArrayObject.h
--- a/js/src/vm/ArrayObject.h   Sat Apr 07 00:55:15 2018 +0300
+++ b/js/src/vm/ArrayObject.h   Sun Apr 08 00:01:23 2018 +0000
@@ -60,6 +60,14 @@
        getElementsHeader()->length = length;
    }
```

```
+    void setCapacityInt32(uint32_t length) {
+        getElementsHeader()->capacity = length;
+    }
+
+    void setInitializedLengthInt32(uint32_t length) {
+        getElementsHeader()->initializedLength = length;
+    }
+
    // Make an array object with the specified initial state.
    static inline ArrayObject*
    createArray(JSContext* cx,
```

应用如下所示的补丁，只需仔细检查它是否已正确应用（不应该出现任何冲突）：

```
>cd gecko-dev\js

gecko-dev\js>git apply c:\work\codes\blazefox\blaze.patch

gecko-dev\js>git diff
diff --git a/js/src/builtin/Array.cpp b/js/src/builtin/Array.cpp
index 1655adbf58..e2ee96dd5e 100644
--- a/js/src/builtin/Array.cpp
+++ b/js/src/builtin/Array.cpp
@@ -202,6 +202,20 @@ GetLengthProperty(JSContext* cx, HandleObject obj, uint64_t* lengthp)
    return ToLength(cx, value, lengthp);
}

+static MOZ_ALWAYS_INLINE bool
+BlazeSetLengthProperty(JSContext* cx, HandleObject obj, uint64_t length)
+{
+    if (obj->is<ArrayObject>()) {
+        obj->as<ArrayObject>().setLengthInt32(length);
+        obj->as<ArrayObject>().setCapacityInt32(length);
+        obj->as<ArrayObject>().setInitializedLengthInt32(length);
+        return true;
+    }
+    return false;
+}
```

此时，您可以安装 Mozilla-Build，它是一个元安装程序，为您提供在 Mozilla
上进行开发所需的所有工具（工具链，各种脚本等）。在撰写本文时，最新的可用版本是版本 3.2，可在此处获得：MozillaBuildSetup-3.2.exe

安装完成后，通过运行start-shell.bat批处理文件启动Mozilla shell。 转到js \ src文件夹中克隆的位置，然后键入以下内容以配置js.exe的x64调试版本：

```
over@compiler /d/gecko-dev/js/src$ autoconf-2.13

over@compiler /d/gecko-dev/js/src$ mkdir build.asserts

over@compiler /d/gecko-dev/js/src$ cd build.asserts

over@compiler /d/gecko-dev/js/src/build.asserts$ ../configure --host=x86_64-pc-mingw32 --target=x86_64-pc-mingw32 --enable-deb
```

用 mozmake 开始编译

```
over@compiler /d/gecko-dev/js/src/build.asserts$ mozmake -j2
```

然后，你应该能够在一个目录中输入 ./js/src/js.exe,./mozglue/build/mozglue.dll 和 ./config/external/nspr/pr/nspr4.dll 并且这里：

```
over@compiler ~/mozilla-central/js/src/build.asserts/js/src
$ js.exe --version
JavaScript-C64.0a1
```

对于优化的构建，您可以通过这种方式调用 configure:

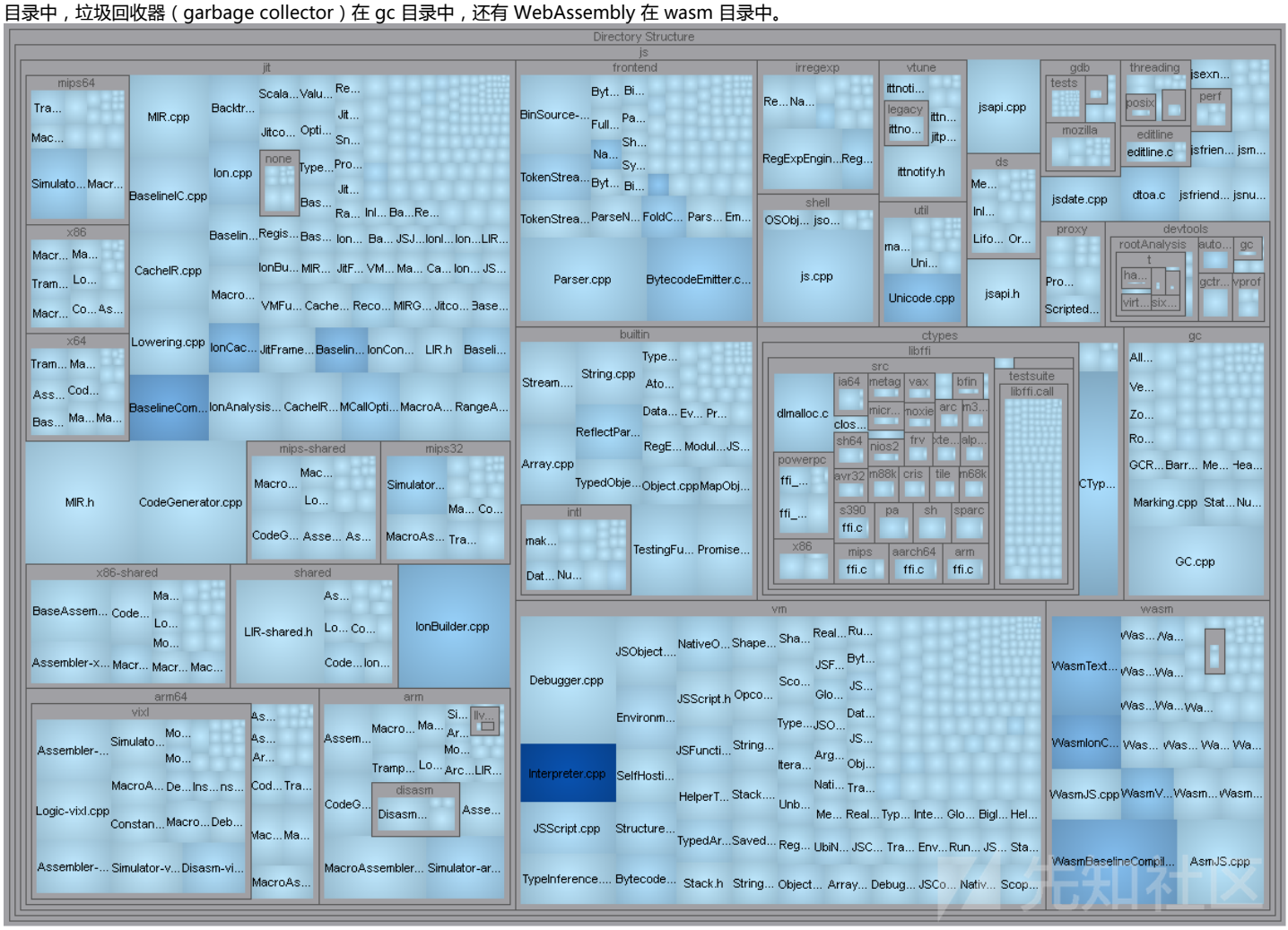
```
over@compiler /d/gecko-dev/js/src/build.opt$ ../configure --host=x86_64-pc-mingw32 --target=x86_64-pc-mingw32 --disable-debug
```

SpiderMonkey

背景

SpiderMonkey 是 Mozilla 的 JavaScript 引擎的名称，它的源代码可以通过 gecko-dev 存储库（在 js 目录下）在 Github 上获得。SpiderMonkey 在 Firefox 使用，更确切地说是由 Web 引擎 Gecko 使用。如果您喜欢它，您甚至可以将解释器嵌入到您自己的第三方应用程序中。
该项目相当大，这里有一些我粗略统计的数据：

- ~3k 个类
- ~576k 行代码
- ~1.2k 个文件
- ~48k 个函数
  正如您在下面的树形图视图中看到的那样（越大，线越多;蓝色越深，圈复杂度越高）js 引擎基本上分为六大部分：JIT编译器引擎名为 Baseline 和 IonMonkey在 Jit 目录中，front-end 在 frontend 目录中，JavaScript virtual-machine 在 VM 目录中，一大堆内置函数在 builtin 目录中，垃圾回收器（garbage collector）在 gc 目录中，还有 WebAssembly 在 wasm 目录中。



我现在看到的大部分内容都存在于vm，builtin和gc文件夹中。对我们来说另一件好事是，还有大量关于SpiderMoneky内部，和设计相关的公共文档等。
以下是我发现有趣的一些链接（有些链接可能已过时，但此时我们只是挖掘我们可以找到的所有公开信息）如果您希望在进一步开发之前获得更多背景知识：

- SpiderMonkeys
- SpiderMonkey Internals
- JSAPI
- GC Rooting guide
- IonMonkey JIT
- The Performance Of Open Source Software: MemShrink

JS::Values and JSObjects

您可能首先想到的是本地 JavaScript
对象如何在内存中布局。让我们创建一个包含几种不同本地类型的脚本文件，并直接从内存中dump出来（不要忘记加载符号表）。这里有一个有用的调试技巧，将断点设置 。 由于您可以将任意 JavaScript 对象传递给函数，因此可以非常轻松地从调试器中检索其地址。 您也可以使用 objectAddress，虽然它只能在 shell 中访问，但有时非常有用。

```
js> a = {}
({})

js> objectAddress(a)
"000002576F8801A0"
```

另一个非常有用的方法是 dumpObject ，但是这个方法只能从 shell 的调试版本中获得：

```
js> a = {doare : 1}
({doare:1})

js> dumpObject(a)
object 20003e8e160
 global 20003e8d060 [global]
 class 7ff624d94218 Object
 lazy group
 flags:
 proto <Object at 20003e90040>
 properties:
   "doare": 1 (shape 20003eb1ad8 enumerate slot 0)
```

还有一些其他可能有趣的实用函数通过 shell 传递给 JavaScript 。如果你想枚举它们，你可以运行Object.getOwnPropertyNames（this）：

```
js> Object.getOwnPropertyNames(this)
["undefined", "Boolean", "JSON", "Date", "Math", "Number", "String", "RegExp", "InternalError", "EvalError", "RangeError", "Ty
```

要在调用 Math.atan2 JavaScript 函数时中断调试器，可以在以下符号上设置断点：

```
0:001> bp js!js::math_atan2
```

现在只需创建一个包含以下内容的 foo.js 文件：

```
'use strict';

const Address = Math.atan2;

const A = 0x1337;
Address(A);

const B = 13.37;
Address(B);

const C = [1, 2, 3, 4, 5];
Address(C);
```

此时您有两个选择：要么将上述脚本加载到 JavaScript shell 中并附加调试器，要么我建议使用 TTD
跟踪程序执行。当您尝试研究复杂的软件时，它会使事情变得如此简单。如果你从未尝试过，现在就去做，你会明白的。
是时候加载跟踪并浏览一下：

```
0:001> g
Breakpoint 0 hit
js!js::math_atan2:
00007ff6`9b3fe140 56              push    rsi

0:000> lsa .
  260: }
  261:
  262: bool
  263: js::math_atan2(JSContext* cx, unsigned argc, Value* vp)
>  264: {
  265:     CallArgs args = CallArgsFromVp(argc, vp);
  266:
  267:     return math_atan2_handle(cx, args.get(0), args.get(1), args.rval());
  268: }
  269:
```

此时，你应该像上面那样中断调试器。 为了能够检查传递的 JavaScript 对象，我们需要了解如何将 JavaScript 参数传递给本机 C++ 函数。
它的工作方式是：vp 是一个大小为 argc + 2 的 JS :: Value 数组的指针（一个用于返回值/调用者，一个用于 this 对象）。函数通常不直接通过 vp
访问数组。而是将它包装在一个 JS :: CallArgs 对象中，该对象抽象出需要计算的 JS :: Value 的数量，并提供其功能，如：JS :: CallArgs :: get，JS :: CallArgs :: rval等。它还抽象出与 GC 相关的操作，以使对象保持活动状态。 所以我们需 dump vp 指向的内存。

```
0:000> dqs @r8 l@rdx+2
0000028f`87ab8198  fffe028f`877a9700
0000028f`87ab81a0  fffe028f`87780180
0000028f`87ab81a8  fff88000`00001337
```

我们注意到的第一件事是每个 Value 对象好像都设置了高8位。
通常，在指针中设置一个hax的标志来编码更多的信息（比如类型，译者注：这高8位实际上是表明了数据的类型，0xfff88000 表示 0x00001337 是个整型数据），因为这部分地址空间无法从 Windows 上的用户模式寻址。
至少我们认识到 0x1337 值是什么。 让我们继续第二次调用 Addressnow。

```
0:000> g
Breakpoint 0 hit
js!js::math_atan2:
00007ff6`9b3fe140 56              push    rsi

0:000> dqs @r8 l@rdx+2
0000028f`87ab8198  fffe028f`877a9700
0000028f`87ab81a0  fffe028f`87780180
0000028f`87ab81a8  402abd70`a3d70a3d

0:000> .formats 402abd70`a3d70a3d
Evaluate expression:
 Hex:     402abd70`a3d70a3d
 Double:  13.37
```

这是另一个我们认识到的常数。这次，整个 quad-word 用于表示 double 数。 最后，这里是传递给"Address"的第三次调用的 Array 对象：

```
0:000> g
Breakpoint 0 hit
js!js::math_atan2:
00007ff6`9b3fe140 56              push    rsi

0:000> dqs @r8 l@rdx+2
0000028f`87ab8198  fffe028f`877a9700
0000028f`87ab81a0  fffe028f`87780180
0000028f`87ab81a8  fffe028f`87790400
```

有趣。如果我们看一下 JS :: Value 结构，看起来 quad-word 的下半部分是指向某个对象的指针。

```
0:000> dt -r2 js::value
  +0x000 asBits_          : Uint8B
  +0x000 asDouble_        : Float
  +0x000 s_               : JS::Value::<unnamed-type-s_>
     +0x000 payload_         : JS::Value::<unnamed-type-s_>::<unnamed-type-payload_>
        +0x000 i32_             : Int4B
        +0x000 u32_             : Uint4B
        +0x000 why_             : JSWhyMagic
```

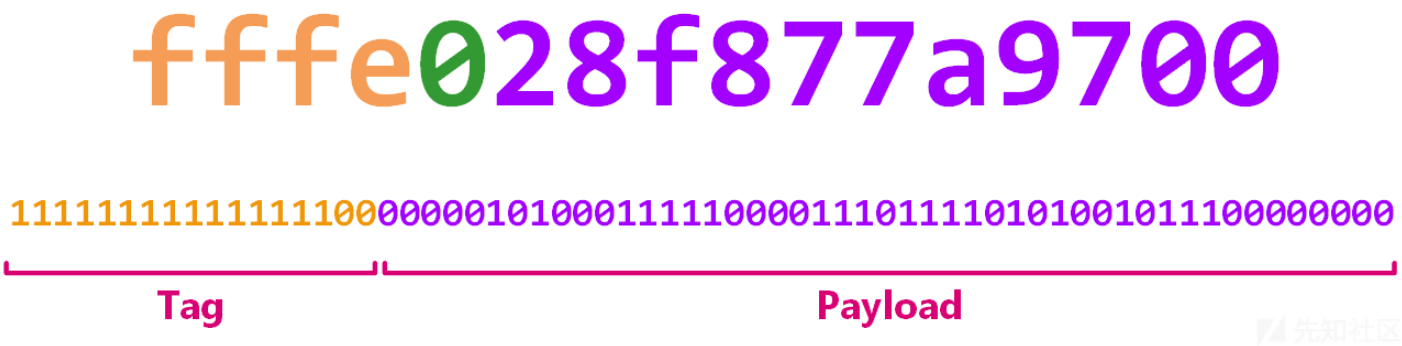
通过查看 public / Value.h，我们很快就能理解上面所看到的内容。JS :: Value 的 17 个高位（在源代码中称为 JSVAL_TAG）用于编码类型信息。较低的 47 位（称为 JSVAL_TAG_SHIFT）是普通类型的值（整数，布尔值等）或指向 JSObject 的指针。 这部分称为 payload_

```
union alignas(8) Value {
 private:
   uint64_t asBits_;
   double asDouble_;

   struct {
       union {
           int32_t i32_;
           uint32_t u32_;
           JSWhyMagic why_;
       } payload_;
```

现在让我们以 JS :: Value 0xfff8800000001337 为例。 要提取它的标记位，我们可以把它右移 47，并提取有payload_
（这里是一个整数，一个普通的类型），我们可以用 2**47 - 1 覆盖它。与上面的数组 JS :: Value相同。

```
In [5]: v = 0xfff8800000001337

In [6]: hex(v >> 47)
Out[6]: '0x1fff1L'

In [7]: hex(v & ((2**47) - 1))
Out[7]: '0x1337L'

In [8]: v = 0xfffe028f877a9700
```

```
In [9]: hex(v >> 47)
Out[9]: '0x1fffcL'

In [10]: hex(v & ((2**47) - 1))
Out[10]: '0x28f877a9700L'
```



上面的 0x1fff1 常量是 JSVAL_TAG_INT32，而 0x1fffc 是 JSValueType 中定义的 JSVAL_TAG_OBJECT，这都是有依据的：

```
enum JSValueType : uint8_t
{
   JSVAL_TYPE_DOUBLE              = 0x00,
   JSVAL_TYPE_INT32               = 0x01,
   JSVAL_TYPE_BOOLEAN             = 0x02,
   JSVAL_TYPE_UNDEFINED           = 0x03,
   JSVAL_TYPE_NULL                = 0x04,
   JSVAL_TYPE_MAGIC               = 0x05,
   JSVAL_TYPE_STRING              = 0x06,
   JSVAL_TYPE_SYMBOL              = 0x07,
   JSVAL_TYPE_PRIVATE_GCTHING     = 0x08,
   JSVAL_TYPE_OBJECT              = 0x0c,

   // These never appear in a jsval; they are only provided as an out-of-band
   // value.
   JSVAL_TYPE_UNKNOWN             = 0x20,
   JSVAL_TYPE_MISSING             = 0x21
};

JS_ENUM_HEADER(JSValueTag, uint32_t)
{
   JSVAL_TAG_MAX_DOUBLE           = 0x1FFF0,
   JSVAL_TAG_INT32                = JSVAL_TAG_MAX_DOUBLE | JSVAL_TYPE_INT32,
   JSVAL_TAG_UNDEFINED            = JSVAL_TAG_MAX_DOUBLE | JSVAL_TYPE_UNDEFINED,
   JSVAL_TAG_NULL                 = JSVAL_TAG_MAX_DOUBLE | JSVAL_TYPE_NULL,
   JSVAL_TAG_BOOLEAN              = JSVAL_TAG_MAX_DOUBLE | JSVAL_TYPE_BOOLEAN,
   JSVAL_TAG_MAGIC                = JSVAL_TAG_MAX_DOUBLE | JSVAL_TYPE_MAGIC,
   JSVAL_TAG_STRING               = JSVAL_TAG_MAX_DOUBLE | JSVAL_TYPE_STRING,
   JSVAL_TAG_SYMBOL               = JSVAL_TAG_MAX_DOUBLE | JSVAL_TYPE_SYMBOL,
   JSVAL_TAG_PRIVATE_GCTHING      = JSVAL_TAG_MAX_DOUBLE | JSVAL_TYPE_PRIVATE_GCTHING,
   JSVAL_TAG_OBJECT               = JSVAL_TAG_MAX_DOUBLE | JSVAL_TYPE_OBJECT
} JS_ENUM_FOOTER(JSValueTag);
```

现在我们知道什么是 JS :: Value，让我们看一下 Array 在内存中的样子，因为这会在之后用到。 重新启动目标并跳过第一个双重中断。

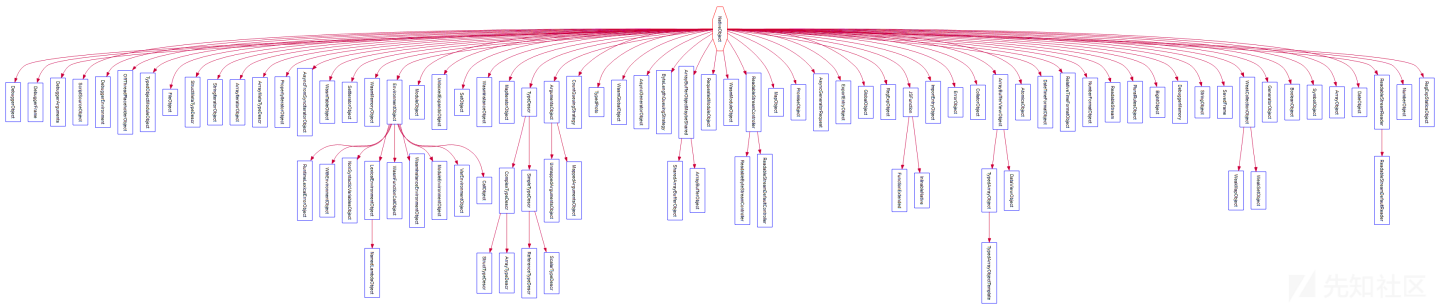
```
0:000> .restart /f

0:008> g
Breakpoint 0 hit
js!js::math_atan2:
00007ff6`9b3fe140 56              push    rsi

0:000> g
Breakpoint 0 hit
js!js::math_atan2:
00007ff6`9b3fe140 56              push    rsi

0:000> g
```

```
Breakpoint 0 hit
js!js::math_atan2:
00007ff6`9b3fe140 56              push    rsi

0:000> dqs @r8 l@rdx+2
0000027a`bf5b8198  fffe027a`bf2a9480
0000027a`bf5b81a0  fffe027a`bf280140
0000027a`bf5b81a8  fffe027a`bf2900a0

0:000> dqs 27a`bf2900a0
0000027a`bf2900a0  0000027a`bf27ab20
0000027a`bf2900a8  0000027a`bf2997e8
0000027a`bf2900b0  00000000`00000000
0000027a`bf2900b8  0000027a`bf2900d0
0000027a`bf2900c0  00000005`00000000
0000027a`bf2900c8  00000005`00000006
0000027a`bf2900d0  fff88000`00000001
0000027a`bf2900d8  fff88000`00000002
0000027a`bf2900e0  fff88000`00000003
0000027a`bf2900e8  fff88000`00000004
0000027a`bf2900f0  fff88000`00000005
0000027a`bf2900f8  4f4f4f4f`4f4f4f4f
```

此时我们认识到数组的内容：从 1 到 5 的数值，作为 JS :: Value
的五个整数。我们也可以看到哪个位置的数据可能是存放的数组大小和容量，但剩下的数据很难猜到存放的是什么。

```
0:000> dt JSObject
  +0x000 group_           : js::GCPtr<js::ObjectGroup *>
  +0x008 shapeOrExpando_  : Ptr64 Void

0:000> dt js::NativeObject
  +0x000 group_           : js::GCPtr<js::ObjectGroup *>
  +0x008 shapeOrExpando_  : Ptr64 Void
  +0x010 slots_           : Ptr64 js::HeapSlot
  +0x018 elements_        : Ptr64 js::HeapSlot

0:000> dt js::ArrayObject
  +0x000 group_           : js::GCPtr<js::ObjectGroup *>
  +0x008 shapeOrExpando_  : Ptr64 Void
  +0x010 slots_           : Ptr64 js::HeapSlot
  +0x018 elements_        : Ptr64 js::HeapSlot
```

JS :: ArrayObject 在 vm / ArrayObject.h 文件中定义，它是 JS :: NativeObject 类的子类（JS :: NativeObject 子类 JS :: ShapedObject，它自然是 JSObject 的子类）。 请注意，它也基本上是每个其他 JavaScript 对象的子类，如下图所示：



SpiderMonkey 中的本地对象基本上由两个组件组成：

- 一个形状对象，用于描述属性，所述对象的类，更多的是在下面（由字段 shapeOrExpando_ 指向）。
- 存储元素或属性值的存储。

让我们看看对象属性如何存储在内存中。

Shapes

如上所述，shape 对象的作用是描述对象具有的各种属性。从概念上讲，您可以将其视为某种哈希表，其中 Key 是属性名称，值是实际存储属性内容的 slot 。
在进一步阅读之前，我建议您观看由 @bmeurer 和 @mathias 撰写的非常简短的演示文稿，描述如何在 JavaScript 引擎中存储属性：JavaScript 引擎基础：形状和内联缓存。 他们清楚地解释这方面的事，它应该有助于理解接下来会发生什么，这也意味着我不需要尽可能多地介绍。
考虑以下 JavaScript 代码：

```
'use strict';

const Address = Math.atan2;

const A = {
   foo : 1337,
   blah : 'doar-e'
};
Address(A);

const B = {
   foo : 1338,
   blah : 'sup'
};
Address(B);

const C = {
   foo : 1338,
   blah : 'sup'
};
C.another = true;
Address(C);
```

将它放在您最喜欢的调试器下面的 shell 中，以便仔细查看此 shape 对象：

```
0:000> bp js!js::math_atan2

0:000> g
Breakpoint 0 hit
Time Travel Position: D454:D
js!js::math_atan2:
00007ff7`76c9e140 56              push    rsi

0:000> ?? vp[2].asBits_
unsigned int64 0xfffe01fc`e637e1c0

0:000> dt js::NativeObject 1fc`e637e1c0 shapeOrExpando_
  +0x008 shapeOrExpando_ : 0x000001fc`e63ae880 Void

0:000> ?? ((js::shape*)0x000001fc`e63ae880)
class js::Shape * 0x000001fc`e63ae880
  +0x000 base_            : js::GCPtr<js::BaseShape *>
  +0x008 propid_          : js::PreBarriered<jsid>
  +0x010 immutableFlags   : 0x2000001
  +0x014 attrs            : 0x1 ''
  +0x015 mutableFlags     : 0 ''
  +0x018 parent           : js::GCPtr<js::Shape *>
  +0x020 kids             : js::KidsPointer
  +0x020 listp            : (null)

0:000> ?? ((js::shape*)0x000001fc`e63ae880)->propid_.value
struct jsid
  +0x000 asBits           : 0x000001fc`e63a7e20
```

在实现中，JS :: Shape 描述了一个属性；它的名称和 slot 号。为了描述它们中的几个，通过父字段（和其他字段）将 shapes 链接在一起。slot 号（稍后用于查找属性内容）存储在 immutableFlags 字段的低位中。属性名称存储为 propid_ 字段中的 jsid 。
我知道现在你面前有很多抽象的信息。但是，让我们一层一层的来分析; 从上面的 shape 开始。此 JS :: Shape 对象描述了一个属性，该值存储在 slot 号为 1（0x2000001 和 SLOT_MASK）中。为了得到它的名字，我们转储它的propid_ 字段，即 0x000001fce63a7e20 。
什么是 jsid ？ jsid 是另一种类型的标记指针，其中类型信息这次以低三位编码。

感谢那些较低的位，我们知道这个地址指向一个字符串，它应该匹配我们的属性名称:)。

```
0:000> ?? (char*)((JSString*)0x000001fc`e63a7e20)->d.inlineStorageLatin1
char * 0x000001fc`e63a7e28
"blah"
```

如上所述， shape 对象链接在一起。 如果我们转储其父级，我们希望找到描述我们的第二个属性 foo 的 shape：

```
0:000> ?? ((js::shape*)0x000001fc`e63ae880)->parent.value
class js::Shape * 0x000001fc`e63ae858
  +0x000 base_            : js::GCPtr<js::BaseShape *>
  +0x008 propid_          : js::PreBarriered<jsid>
  +0x010 immutableFlags   : 0x2000000
  +0x014 attrs            : 0x1 ''
  +0x015 mutableFlags     : 0x2 ''
  +0x018 parent           : js::GCPtr<js::Shape *>
  +0x020 kids             : js::KidsPointer
  +0x020 listp            : 0x000001fc`e63ae880 js::GCPtr<js::Shape *>

0:000> ?? ((js::shape*)0x000001fc`e63ae880)->parent.value->propid_.value
struct jsid
  +0x000 asBits           : 0x000001fc`e633d700

0:000> ?? (char*)((JSString*)0x000001fc`e633d700)->d.inlineStorageLatin1
char * 0x000001fc`e633d708
"foo"
```

按 g 继续执行并检查第二个对象是否共享相同的形状层次结构（0x000001fce63ae880）：

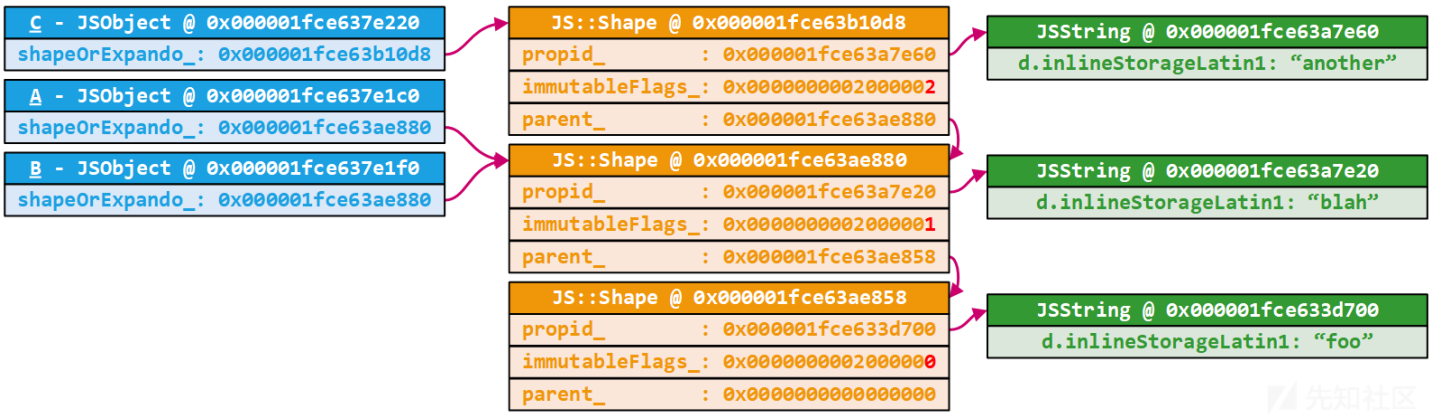
```
0:000> g
Breakpoint 0 hit
Time Travel Position: D484:D
js!js::math_atan2:
00007ff7`76c9e140 56              push    rsi

0:000> ?? vp[2].asBits_
unsigned int64 0xfffe01fc`e637e1f0

0:000> dt js::NativeObject 1fc`e637e1f0 shapeOrExpando_
  +0x008 shapeOrExpando_ : 0x000001fc`e63ae880 Void
```

正如预期的那样，B 确实会分享它，即使 A和 B 存储不同的属性值。 关注我们现在向 C 添加另一个属性时会发生什么？ 要查找，请最后按 g 一次：

```
0:000> g
Breakpoint 0 hit
Time Travel Position: D493:D
js!js::math_atan2:
00007ff7`76c9e140 56              push    rsi

0:000> ?? vp[2].asBits_
union JS::Value
  +0x000 asBits_          : 0xfffe01e7`c247e1c0

0:000> dt js::NativeObject 1fc`e637e1f0 shapeOrExpando_
  +0x008 shapeOrExpando_ : 0x000001fc`e63b10d8 Void
```

```
0:000> ?? ((js::shape*)0x000001fc`e63b10d8)
class js::Shape * 0x000001fc`e63b10d8
  +0x000 base_            : js::GCPtr<js::BaseShape *>
  +0x008 propid_          : js::PreBarriered<jsid>
  +0x010 immutableFlags   : 0x2000002
  +0x014 attrs            : 0x1 ''
  +0x015 mutableFlags     : 0 ''
  +0x018 parent           : js::GCPtr<js::Shape *>
  +0x020 kids             : js::KidsPointer
  +0x020 listp            : (null)

0:000> ?? ((js::shape*)0x000001fc`e63b10d8)->propid_.value
struct jsid
  +0x000 asBits           : 0x000001fc`e63a7e60

0:000> ?? (char*)((JSString*)0x000001fc`e63a7e60)->d.inlineStorageLatin1
char * 0x000001fc`e63a7e68
"another"

0:000> ?? ((js::shape*)0x000001fc`e63b10d8)->parent.value
class js::Shape * 0x000001fc`e63ae880
```

新的 JS :: Shape 被分配（0x000001e7c24b1150），其父级是前一组形状（0x000001e7c24b1150）。 有点像在链表中添加节点。



Slots

在上一节中，我们讨论了很多关于属性名称如何存储在内存中的问题。 那属性的值在哪里呢？

为了回答这个问题，我们抛出了我们在调试器中获得的先前 TTD 跟踪，并在第一次调用 Math.atan2 时返回：

```
Breakpoint 0 hit
Time Travel Position: D454:D
js!js::math_atan2:
00007ff7`76c9e140 56              push    rsi

0:000> ?? vp[2].asBits_
unsigned int64 0xfffe01fc`e637e1c0
```

因为我们已经 dump 了描述 foo 和 blah 属性的 js :: Shape 对象的过程，所以我们知道它们的属性值分别存储在slot 0 和 slot 1 中。 为了查看这些，我们只是在 js :: NativeObject 之后 dump 内存：

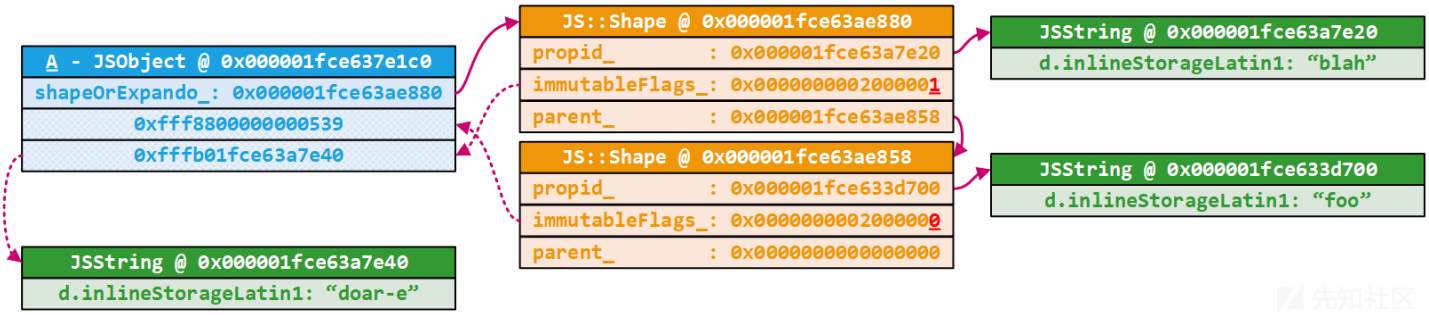
```
0:000> ?? vp[2].asBits_
unsigned int64 0xfffe01fc`e637e1c0
0:000> dt js::NativeObject 1fce637e1c0
  +0x000 group_           : js::GCPtr<js::ObjectGroup *>
  +0x008 shapeOrExpando_  : 0x000001fc`e63ae880 Void
  +0x010 slots_           : (null)
  +0x018 elements_        : 0x00007ff7`7707dac0 js::HeapSlot

0:000> dqs 1fc`e637e1c0
000001fc`e637e1c0  000001fc`e637a520
000001fc`e637e1c8  000001fc`e63ae880
000001fc`e637e1d0  00000000`00000000
000001fc`e637e1d8  00007ff7`7707dac0 js!emptyElementsHeader+0x10
000001fc`e637e1e0  fff88000`00000539 <- foo
```

```
000001fc`e637e1e8  fffb01fc`e63a7e40 <- blah
```

当然，第二个属性是指向 JSString 的另一个 js :: Value，我们也可以 dump 它：

```
0:000> ?? (char*)((JSString*)0x1fce63a7e40)->d.inlineStorageLatin1
char * 0x000001fc`e63a7e48
"doar-e"
```

下面是一个描述对象层次结构的图表，以清除任何可能的混淆：



到目前，我想要描述的内容都已经被涵盖的差不多了，它应该足以帮助我们理解接下来的内容。你可以使用此背景检查大多数 JavaScript 对象。我遇到的唯一“odd-balls”类型是存储长度属性的 JavaScript 数组，例如在 js :: ObjectElements 对象中; 但这就是它。

```
0:000> dt js::ObjectElements
  +0x000 flags            : Uint4B
  +0x004 initializedLength : Uint4B
  +0x008 capacity         : Uint4B
  +0x00c length           : Uint4B
```


上一篇：某php 远程代码执行审计 下一篇：TP Link SR20 ACE漏洞分析

1. 0 条回复
   - 动动手指，沙发就是你的了！