SpiderMonkey 中的一些数据结构

f0**** / 2019-08-25 09:40:00 / 浏览数 3070 安全技术 二进制安全 顶(0) 踩(0)

注：本篇文章是翻译文章，主要内容是讲解 SpiderMonkey 中的数据结构。链接：https://vigneshsrao.github.io/play-with-spidermonkey/

前言

在开始之前，我想说很多内容来自参考文献，这篇文章或多或少是关于我摆弄那里提到的内容。

构建 SpiderMonkey

要调试 SpiderMonkey，您可能需要先构建一个js shell。 JS shell 基本上是一个 js 解释器。 可以在此处找到构建说明。 我将其包括在内以供参考。

```
hg clone http://hg.mozilla.org/mozilla-central spidermonkey

cp configure.in configure && autoconf2.13
mkdir build_DBG.OBJ
cd build_DBG.OBJ
../configure --disable-debug --disable-optimize #
make ## or make -j8
cd dist/bin/
./js
```

PS：我第一次在 brucechen的文章中看到了这个。
注意：我正在禁用调试选项，因为这将添加许多断点，将破坏我们的漏洞，一旦我们到达那个部分，但如果你只是在尝试调试
SpiderMonkey，那么你应该启用它。

Representing Values

本节的大部分内容都基于这篇 phrack 文章。 作者非常清楚地解释了一切，绝对值得一读。

JSValue

在JavaScript中，我们可以为变量赋值而不实际定义它们的“类型”。所以，我们可以这样做 a = "this is a string" or a=1234" ，而不需要像 C 语言中一样指明 int a ， char a。那么 JS 如何跟踪变量的数据类型呢？
这里，所有“类型”的数据都表示为 JS:Value 的对象。JS:Value/jsval 通过在一个单元中编码“type”和“value”来标示各种类型。
在 jsval 中，前17位用于表示 jsval 类型的标记。低 47 位用于实际值。
让我们看一个例子。 运行js shell并创建一个create数组来保存不同类型的值。

```
js>  a=[0x11223344, "STRING", 0x44332211, true]
[287454020, "STRING", 1144201745, true]
```

所以我们的数组就像 - [int，string，int，Boolean]。 现在让我们将附加gdb调试，并查看它们在内存中的形式。

```
gdb -p $(pidof js)

gdb-peda$ find 0x11223344 # Searching for the array - all elements will lie consecutively
Searching for '0x11223344' in: None ranges
Found 1 results, display max 1 items:
mapped : 0x7f8e531980d0 --> 0xfff8800011223344

gdb-peda$ x/4xg 0x7f8e531980d0
0x7f8e531980d0: 0xfff8800011223344  0xfffb7f8e531ae6a0
0x7f8e531980e0: 0xfff8800044332211  0xfff9000000000001
```

所以 int 0x11223344 存储为 0xfff8800011223344 。 以下是 js / public / Value.h 中的相关代码。

```
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);
```

```
enum JSValueShiftedTag : uint64_t
{
   JSVAL_SHIFTED_TAG_MAX_DOUBLE      = ((((uint64_t)JSVAL_TAG_MAX_DOUBLE)     << JSVAL_TAG_SHIFT) | 0xFFFFFFFF),
   JSVAL_SHIFTED_TAG_INT32           = (((uint64_t)JSVAL_TAG_INT32)           << JSVAL_TAG_SHIFT),
   JSVAL_SHIFTED_TAG_UNDEFINED       = (((uint64_t)JSVAL_TAG_UNDEFINED)       << JSVAL_TAG_SHIFT),
   JSVAL_SHIFTED_TAG_NULL            = (((uint64_t)JSVAL_TAG_NULL)            << JSVAL_TAG_SHIFT),
   JSVAL_SHIFTED_TAG_BOOLEAN         = (((uint64_t)JSVAL_TAG_BOOLEAN)         << JSVAL_TAG_SHIFT),
   JSVAL_SHIFTED_TAG_MAGIC           = (((uint64_t)JSVAL_TAG_MAGIC)           << JSVAL_TAG_SHIFT),
   JSVAL_SHIFTED_TAG_STRING          = (((uint64_t)JSVAL_TAG_STRING)          << JSVAL_TAG_SHIFT),
   JSVAL_SHIFTED_TAG_SYMBOL          = (((uint64_t)JSVAL_TAG_SYMBOL)          << JSVAL_TAG_SHIFT),
   JSVAL_SHIFTED_TAG_PRIVATE_GCTHING = (((uint64_t)JSVAL_TAG_PRIVATE_GCTHING) << JSVAL_TAG_SHIFT),
   JSVAL_SHIFTED_TAG_OBJECT          = (((uint64_t)JSVAL_TAG_OBJECT)          << JSVAL_TAG_SHIFT)
};
```

代码很容易理解

- 每个类型（Int，String，Boolean等）由枚举 JSValueType 中显示的数字表示
- 这与 JSVAL_TAG_MAX_DOUBLE 一致，如枚举 JSValueTag 中所示。 这个或那个值实际上是将在最终表示中使用的"标签"。
- 通过右移47位，将17位标记设为64位。

所以 int 的标签就是
(1 | 0x1FFF0) « 47 = 0xfff8800000000000

实际 int 的值与此标记一起使用，并在内存中存储为 "0xfff8800011223344" 。

JSObject

上述，就是标记值。JavaScript 也有各种类型的"对象"，如数组。 对象倾向于具有"属性"。

```
obj = { p1: 0x11223344, p2: "STRING", p3: true, p4: [1.2,3.8]};
```

在上面的例子中，p1，p2，p3 和 p4 是对象 obj 的"属性"。 它们就像 python 词典。 每个属性都有一个映射到它的值。
这可以是任何类型，int，string，Boolean，object 等。这些对象在内存中表示为 JSObject 类的对象。
以下是 NativeObject 类的抽象，它继承了其他类中的 JSObject。

```
class NativeObject
{
   js::GCPtrObjectGroup group_;
   void* shapeOrExpando_;
   js::HeapSlot *slots_;
   js::HeapSlot *elements_;
};
```

让我们更详细地讨论这些。

group_

我没有完全理解 group 成员的要求和使用，但我确实在 js / src / vm / JSObject.h 中遇到了以下注释。

> | group_ | member存储对象的组，其中包含其原型对象，其类及其属性的可能类型。
> group 成员本质上是 ObjectGroup 类的成员，具有以下成员。

```
/* Class shared by objects in this group. */
 const Class* clasp_; // set by constructor

 /* Prototype shared by objects in this group. */
 GCPtr<TaggedProto> proto_; // set by constructor

 /* Realm shared by objects in this group. */
 JS::Realm* realm_;; // set by constructor

 /* Flags for this group. */
 ObjectGroupFlags flags_; // set by constructor

 // If non-null, holds additional information about this object, whose
 // format is indicated by the object's addendum kind.
 void* addendum_ = nullptr;

 Property** propertySet = nullptr;
```

> 注释或多或少地解释了每个字段的使用，但是让我进入 clasp 成员，因为它提供了有趣的利用目标。

与注释说的一样，这定义了属于该组的所有对象共享的 JSClass，可用于标识该组。 我们来看看Class 结构。

```
struct MOZ_STATIC_CLASS Class
{
   JS_CLASS_MEMBERS(js::ClassOps, FreeOp);
   const ClassSpec* spec;
   const ClassExtension* ext;
   const ObjectOps* oOps;
   :
   :
}
```

除了 ClassOps 之外，我对其他所有属性都不太了解，但是我会在这里更新它。 ClassOps
基本上是一个指向结构的指针，该结构包含许多函数指针，这些函数指针定义对象的特定操作是如何发生的。 我们来看看这个 ClassOps 结构。

```
struct MOZ_STATIC_CLASS ClassOps
{
   /* Function pointer members (may be null). */
   JSAddPropertyOp     addProperty;
   JSDeletePropertyOp  delProperty;
   JSEnumerateOp       enumerate;
   JSNewEnumerateOp    newEnumerate;
   JSResolveOp         resolve;
   JSMayResolveOp      mayResolve;
   FinalizeOp          finalize;
   JSNative            call;
   JSHasInstanceOp     hasInstance;
   JSNative            construct;
   JSTraceOp           trace;
};
```

例如，addProperty 字段中的函数指针定义了在调用新属性时要调用的函数。 在这篇文章中，很好地解释了这一切，特别是对于那些从 SpiderMonkey 开发的人而言，这是一篇非常好的文章。 回到这一点，这里有一个函数指针数组。
如果我们设法覆盖它们中的任何一个，就可以将任意写入转换为任意代码执行。

但这不是那么容易。 问题是这个包含函数指针的区域是一个 r-x 区域（没有写权限）。 但是，只要我们有任意写入，我们就可以轻松伪造整个 ClassOps 结构，并用指向伪结构的指针覆盖指向组字段中实际 ClassOps 的指针。

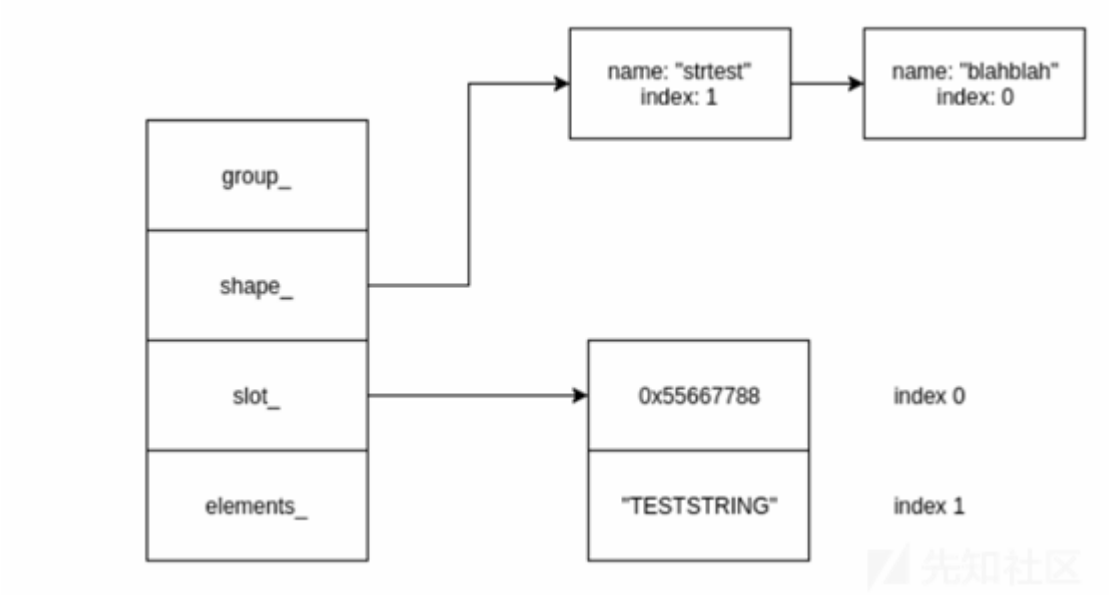
因此，只要我们有任意写入，我们就有了获取代码执行的方法。

shape_ and slots_

那么 js 如何跟踪对象的属性呢？ 请考虑以下代码段。

```
obj = {}
obj.blahblah = 0x55667788
obj.strtest = "TESTSTRING"
```

obj是一个数组，但它也有一些属性。 现在我们必须跟踪属性名称及其值。 为此，它使用对象的shape_ 和 slots_ 字段。 slots_ 字段是包含与每个属性关联的值的字段。 它基本上是一个只包含值（无名称）的数组。 shape_ 包含属性的名称以及 slots_ array 的索引，其中将显示此属性的值。
也许以下图片比我解释的更直观:)



好了，接下来让我们看看gdb调试下的内存情况。

```
gdb-peda$ x/4xg 0x7f7f01b90120
0x7f7f01b90120: 0x00007f7f01b8a310  0x00007f7f01bb18d0 ----> shape_
0x7f7f01b90130: 0x00007f7f01844ec0  0x000000000174a490
                      |
                      +----------------------------------> slots_

gdb-peda$ tel 0x00007f7f01bb18d0 4
0000| 0x7f7f01bb18d0 --> 0x7f7f01b8b0e0 --> 0x2a26380 (:PlainObject::class_>:   0x000000000162a4bf)
0008| 0x7f7f01bb18d8 --> 0x7f7f01bae6c0 --> 0x70000004a  # Property Name
0016| 0x7f7f01bb18e0 --> 0xfffe000100000001 # Index in slots_ array is '1' (last 3 bytes)
0024| 0x7f7f01bb18e8 --> 0x7f7f01bb18a8 --> 0x7f7f01b8b0e0 --> 0x2a26380 (:PlainObject::class_>:    0x000000000162a4bf)
                            |
                            +-----> pointer to the next shape

# Looking at the property name.

gdb-peda$ x/2wx 0x7f7f01bae6c0
0x7f7f01bae6c0: 0x0000004a  0x00000007 # metadata of the string. 0x4a is flag I think and 7 is the length of string.
gdb-peda$ x/s
0x7f7f01bae6c8: "strtest" # The last property added, is at the head of the linked list.

# The next pointer

gdb-peda$ tel 0x7f7f01bb18a8 4
0000| 0x7f7f01bb18a8 --> 0x7f7f01b8b0e0 --> 0x2a26380 (:PlainObject::class_>:   0x000000000162a4bf)
0008| 0x7f7f01bb18b0 --> 0x7f7f01bae6a0 --> 0x80000004a
0016| 0x7f7f01bb18b8 --> 0xfffe000102000000
0024| 0x7f7f01bb18c0 --> 0x7f7f01b8cb78 --> 0x7f7f01b8b0e0 --> 0x2a26380 (:PlainObject::class_>:    0x000000000162a4bf)

# Name of the property

gdb-peda$ x/xg 0x7f7f01bae6a0
0x7f7f01bae6a0: 0x000000080000004a
gdb-peda$ x/s
0x7f7f01bae6a8: "blahblah"
```

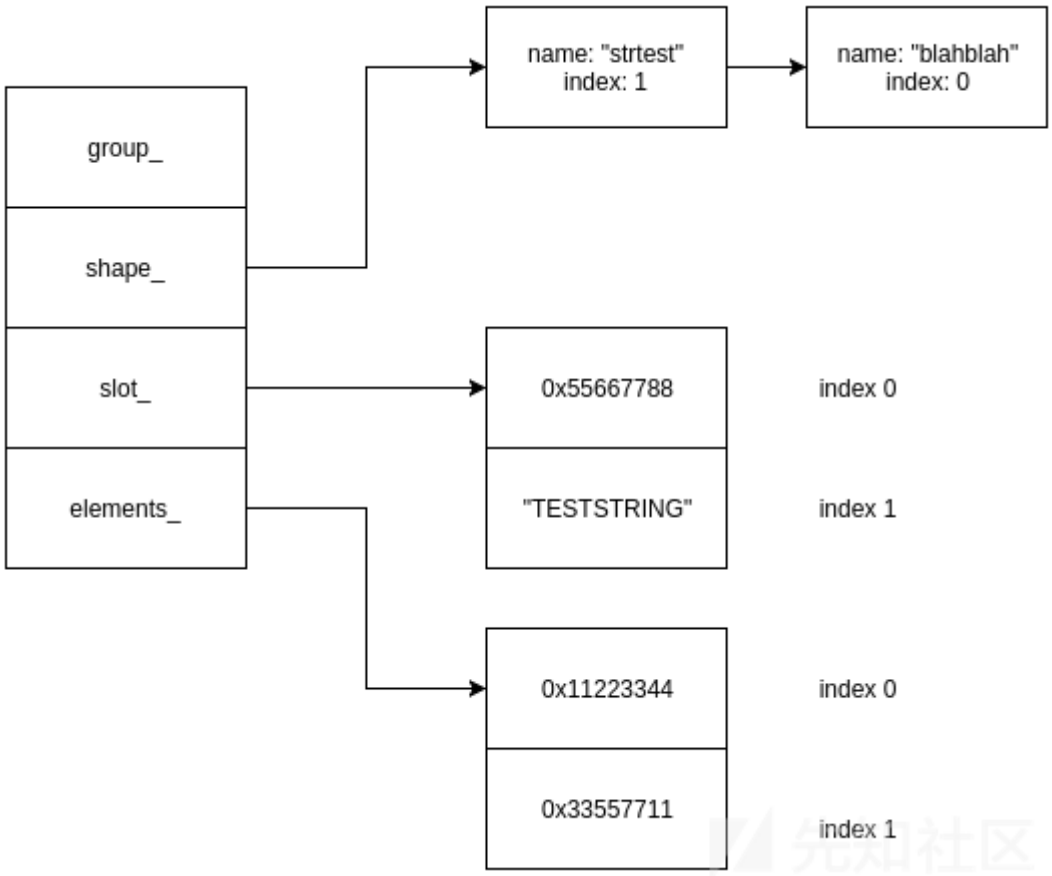
```
# The slots_ array

gdb-peda$ x/xg 0x00007f7f01844ec0
0x7f7f01844ec0: 0xfff8800055667788 # index 0 which is value for the property "blahblah"
0x7f7f01844ec8: 0xfffb7f7f01bae6e0 # index 1 which is value for the property "strtest". This is a string object.

# Dereference index 1, which is a pointer to 0x7f7f01bae6e0

gdb-peda$ x/xg 0x7f7f01bae6e0
0x7f7f01bae6e0: 0x0000000a0000004a
gdb-peda$ x/s
0x7f7f01bae6e8: "TESTSTRING"
```

elements_

在上一节中介绍的示例中，该对象只有一些属性。 如果它也有元素怎么办？ 让我们添加到上面的代码片段

```
obj[0]=0x11223344
obj[1]=0x33557711
```

元素将存储在 elements_ member 指向的数组中。 让我们看看修改后的图像



在 gdb 调试下。

```
# This time we have all previous pointers plus a pointer to the elements_ array

gdb-peda$ x/4xg 0x7f7f01b90120
0x7f7f01b90120: 0x00007f7f01b8a310  0x00007f7f01bb18d0
0x7f7f01b90130: 0x00007f7f01844ec0  0x00007f7f01844f90 ---> elements_

# The array -

gdb-peda$ x/xg 0x00007f7f01844f90
0x7f7f01844f90: 0xfff8800011223344  # index 0
0x7f7f01844f98: 0xfff8800033557711  # index 0
```

现在我们看到我们可以向对象添加任意数量的元素。 所以 elements array 有一个 metadata 成员来跟踪这些元素（这实际上是显式地转换为 ObjectElements 。更详细的信息在 js / src / vm / NativeObject.h ）。 以下是构成 metadata ：

```
uint32_t flags;

/*
* Number of initialized elements. This is <= the capacity, and for arrays
* is <= the length. Memory for elements above the initialized length is
* uninitialized, but values between the initialized length and the proper
* length are conceptually holes.
*/
uint32_t initializedLength;

/* Number of allocated slots. */
uint32_t capacity;

/* 'length' property of array objects, unused for other objects. */
uint32_t length;
```

上面的代码来自 NativeObject.h 中 ObjectElements 的定义。 注释应该说的比较清楚了。 让我们向 obj 对象添加更多元素。

```
obj[2]="asdfasdf"
obj[3]=6.022
```

在 gdb 调试下的情况。

```
gdb-peda$ x/4xg 0x7f7f01b90120
0x7f7f01b90120: 0x00007f7f01b8a310  0x00007f7f01bb18d0
0x7f7f01b90130: 0x00007f7f01844ec0  0x00007f7f01844f90

# size of the metadata is 0x10 bytes

gdb-peda$ x/4wx 0x00007f7f01844f90-0x10
                    Flags       init_len      capacity      length
0x7f7f01844f80: 0x00000000  0x00000004  0x00000006  0x00000000

gdb-peda$ x/4xg
0x7f7f01844f90: 0xfff8800011223344  0xfff8800033557711
0x7f7f01844fa0: 0xfffb7f7f01bae720  0x401816872b020c4a
```

Typed Arrays

译者注：这个标题有点不好解释，可以这样理解。把 ArrayBuffer 初始化为 Uint32Array ， Uint8Array 等其他类似的数组对象。这些数组对象就是 Typed Arrays。

MDN 上的解释

- ArrayBuffer 是一种数据类型，用于表示通用的固定长度二进制数据缓冲区。 你不能直接操纵 ArrayBuffer 的内容;
  相反，可以创建一个类型化的数组视图或一个表示特定格式的缓冲区的 DataView，并使用它来读取和写入缓冲区的内容。

NativeObject 的所有属性都由 ArrayBufferObject 继承。 另外， ArrayBufferObject 具有以下内容

- Pointer to data：以 “private” 形式指向 ArrayBuffer 的数据缓冲区的数据指针。
- length：缓冲区的大小。
- First View：指向引用当前 ArrayBuffer 的第一个视图的指针。
- flags

以 Private 形式存在的指向数据缓冲区的指针，下面从 setPrivate 开始。

```
void setPrivate(void* ptr) {
   MOZ_ASSERT((uintptr_t(ptr) & 1) == 0);
#if defined(JS_NUNBOX32)
   s_.tag_ = JSValueTag(0);
   s_.payload_.ptr_ = ptr;
#elif defined(JS_PUNBOX64)
   asBits_ = uintptr_t(ptr) >> 1;
#endif
   MOZ_ASSERT(isDouble());
}
```

简化一下就是：

```
void setPrivate(void* ptr) {
   asBits_ = uintptr_t(ptr) >> 1;
}
```

注意，它被右移1.（我们将在gdb调试中检查出来）

现在让我们创建一个 ArrayBuffer 并将视图添加到此缓冲区。

```
arrbuf = new ArrayBuffer(0x100);        // ArrayBuffer of size 0x100 bytes.
uint32view = new Uint32Array(arrbuf);   // Adding a Uint32 view.
uint16view = new Uint16Array(arrbuf);   // Adding another view - this time a Uint16 one.
uint32view[0]=0x11223344                // Initialize the buffer with a value.

uint32view[0].toString(16)
// Outputs "11223344"

/* Lets check the Uint16Array */

uint16view[0].toString(16)
// Outputs "3344"

uint16view[1].toString(16)
// Outputs "1122"
```

对同一缓冲区的不同视图允许我们以不同的方式查看缓冲区中的数据。
除 NativeObject 之外，类似 ArrayBuffer 的 TypedArray 还具有以下额外属性。

- Underlying ArrayBuffer：指向 ArrayBuffer 的指针，该 ArrayBuffer 保存此类型数组的数据
- length：数组的长度。 如果 ArrayBuffer 是有 0x20 字节的 Uint32Array 类型，则 length = 0x20 / 4 = 8
- offset
- pointer to data：这是指向原始形式的数据缓冲区的指针，用于增强性能。

让我们开始研究如何在内存中表示所有这些东西。

```
gdb-peda$ x/8xg 0x7f618109a080
0x7f618109a080: 0x00007f618108a8b0 (group_)       0x00007f61810b1a38 (shape_)
0x7f618109a090: 0x0000000000000000 (slots_)       0x000000000174a490 (elements_)
0x7f618109a0a0: 0x00003fb0c0d34b00 (data pointer) 0xfff8800000000100 (length)
0x7f618109a0b0: 0xfffe7f6183d003a0 (first view)   0xfff8800000000008 (flags)

# The data pointer
gdb-peda$ p/x 0x00003fb0c0d34b00 << 1
$2 = 0x7f6181a69600

# The buffer
gdb-peda$ x/2xg 0x7f6181a69600
0x7f6181a69600: 0x0000000011223344  0x0000000000000000

# The Uint32 Array

gdb-peda$ x/8xg 0x7f6183d003a0
0x7f6183d003a0: 0x00007f618108aa30                  0x00007f61810b4a60
0x7f6183d003b0: 0x0000000000000000                  0x000000000174a490
0x7f6183d003c0: 0xfffe7f618109a080 (ArrayBuffer)    0xfff8800000000040 (length)
0x7f6183d003d0: 0xfff8800000000000 (offset)         0x00007f6181a69600 (Pointer to data buffer)

# The Uint16 Array

gdb-peda$ x/8xg 0x7f6183d003e0
0x7f6183d003e0: 0x00007f618108aaf0                  0x00007f61810b4ba0
0x7f6183d003f0: 0x0000000000000000                  0x000000000174a490
0x7f6183d00400: 0xfffe7f618109a080 (ArrayBuffer)    0xfff8800000000080 (length)
0x7f6183d00410: 0xfff8800000000000 (offset)         0x00007f6181a69600 (Pointer to data buffer)
```

由于 TypedArrays 中的数据在保存的时候，没有
“nan-boxing”和C语言数据的类型(译者注，比如C语言有Int，char...）情况，因此在写利用时这就很有用，能满足我们需要从任意位置读取和写入数据的操作。
(译者注：NaN-boxing ，其实是表示无效的double数，具体解释点这里。）
假设，我们现在可以控制 ArrayBuffer 的数据指针。 因此，通过为 Uint32Array 分配损坏的 ArrayBuffer，您可以一次读取和写入 4
个字节，来用于任意位置的数据读写。(译者注：Uint32Array 读取数据是无符号的32位数据，就相当于直接能读取地址，不会返回 NAN 或者其他情况)

那么，相反的情况，如果我们使用普通数组，则从任意位置读取的数据将处于浮点状态，并将数据写入我们需要以浮点形式写入。

译者注： 这部分是用来说明一个事情，就是使用 Uint32Array 类似的初始化 Arraybuffer
之后，我们从内存中读取数据，写入数据时，读到或写入的数据是真正在内存中存在的形式。举例：

```
■■■■
■■        ■■
00001000    0x1CD01234
■■■ Uin32Array ■■■ 0x00001000 ,■■■■■■■■ 0x1CD01234 ■
■■■■■■■■ Array ■■■ 0x00001000■■■■■■■■■■■■■■■■■■■■■■ NAN ■■■■■■■■■ nan-boxing ■
```

结语

所以，总结一下到目前为止学到的东西:)。
当有更多的空闲时间时，我打算为blazefox写一篇文章，这是一个挑战，也是一个非常好的尝试开始浏览器相关的开发。
我知道这篇文章仍然不完整，可能也有错误。 如果您发现其中的任何错误，告诉我，我很乐意纠正它。

参考

- OR'LYEH? The Shadow over Firefox by argp.
- Learning browser exploitation via 33C3 CTF feuerfuchs challenge
- Building SpiderMonkey