蝴蝶效应与程序错误

一个“渣洞”的利用之旅

----by @oldfresher @360 Alpha Team

1. 介绍

一只南美洲亚马孙河流域热带雨林中的蝴蝶，偶尔扇动几下翅膀，可能在美国德克萨斯引起一场龙卷风吗？这我不能确定，我能确定的是程序中的任意一个细微错误经过放大后都可能对程序产生灾难性的后果。在 11 月韩国首尔举行的 PwnFest 比赛中，我们利用了 V8 的一个逻辑错误(CVE-2016-9651)来实现 Chrome 的远程任意代码执行，这个逻辑错误非常微小，可以说是一个品相比较差的渣洞，但通过组合一些奇技淫巧，我们最终实现了对这个漏洞的稳定利用。这个漏洞给我的启示是：“绝不要轻易放弃一个漏洞，绝不要轻易判定一个漏洞不可利用”。

本文将按如下结构进行组织：第二节介绍 V8 引擎中”不可见的”对象私有属性；第三节将引出我们所利用的这个细微的逻辑错误；第四节介绍如何将这个逻辑错误转化为一个越界读的漏洞；第五节会介绍一种将越界读漏洞转化为越界写漏洞的思路，这一节是整个利用流程中最巧妙的一环；第六节是所有环节中最难的一步，详述如何进行全内存空间风水及如何将越界写漏洞转化为任意内存地址读写；第七节介绍从任意内存地址读写到任意代码执行。

2. 隐形的私有属性

在 JavaScript 中，对象是一个关联数组，也可以看做是一个键值对的集合。这些键值对也被称为对象的属性。属性的键可以是字符串也可以是符号，如下所示：

```
var normalObject = {};
normalObject ["string"] = "string";
normalObject [Symbol("d")] = 0.1;
```

代码片段 1：对象属性

上述代码片段先定义了一个对象 **normalObject**，然后给这个对象增加了两个属性。这种可以通过 JavaScript 读取和修改的属性我把它们称作公有属性。可以通过 JavaScript 的 Object 对象提供的两个方法得到一个对象的所有公有属性的键，如下 JavaScript 语句可以得到代码 1 中 **normalObject** 对象的所有公有属性的键。

```
var ownNames = Object.getOwnPropertyNames(normalObject);
var ownSymbols = Object.getOwnPropertySymbols(normalObject);
var ownKeys = ownNames.concat(ownSymbols)
```

执行结果：ownPublicKeys 的值为["string", Symbol(d)]

在 V8 引擎中，除公有属性外，还有一些特殊的 JavaScript 对象存在一些特殊的属性，这些属性只有引擎可以访问，对于用户 JavaScript 则是不可见的，我将这种属性称作私有属性。在 V8 引擎中，符号(Symbol)也包括两种，公有符号和私有符号，公有符号是用户 JavaScript 可以创建和使用的，私有符号则只有引擎可以创建，仅供引擎内部使用。私有属性通常使用私有符号作为键，因为用户 JavaScript 不能得到私有符号，所有也不能以私有符号为键访问私有属性。既然私有属性是隐形的，那如何才能观察到私有属性呢？d8 是 V8 引擎的 Shell 程序，通过 d8 调用运行时函数 DebugPrint 可以查看一个对象的所有属性。比如我们可以通过如下方法查看代码 1 中定义的对 **normalObject** 的所有属性：

```
ggong@ggong-pc:~/ssd1/v8/out/ia32.debug$ ./d8 --allow-natives-syntax
d8> var normalObject  = {};
d8> normalObject ["string"] = "string";
d8> normalObject [Symbol("d")] = 0.1;
d8> %DebugPrint(normalObject )
DebugPrint: 0x30587431: [JS_OBJECT_TYPE]
 - map = 0x53d091c9 [FastProperties]
 - prototype = 0x25605175
 - elements = 0x45384125 <FixedArray[0]> [FAST_HOLEY_ELEMENTS]
 - properties = {
   #string: 0x45384f35 <String[6]: string> (data field at offset 0)
   0x2561ac51 <Symbol: d>: 0x30588d49 <MutableNumber: 0.1> (data field at offset 1)
 }
```

从上示 d8 输出结果可知，**normalObject** 仅有两个公有属性，没有私有属性。现在我们来查看一个特殊对象错误对象的属性情况。

```
d8> var specialObject = new Error("test");
d8> var ownNames = Object.getOwnPropertyNames(specialObject);
d8> var ownSymbols = Object.getOwnPropertySymbols(specialObject);
d8> var ownKeys = ownNames.concat(ownSymbols)undefined
d8> ownKeys
["stack", "message"]                  ---------> all public properties got by normal JavaScript
d8> %DebugPrint(specialObject)
DebugPrint: 0x3058e8cd: [JS_ERROR_TYPE]
 - map = 0x53d0945d [FastProperties]
 - prototype = 0x2560b9e1
 - elements = 0x45384125 <FixedArray[0]> [FAST_HOLEY_SMI_ELEMENTS]
 - properties = {                     ---------> all properties got by DebugPrint
   #stack: 0x453d012d <AccessorInfo> (accessor constant)
```

```
  #message: 0x453bb18d <String[4]: test> (data field at offset 0)
  0x453859f1 <Symbol: stack_trace_symbol>: 0x3058e9c1 <JS Array[6]> (data field at offset
1)
 }
```

对比一下 **specialObject** 对象的公有属性和所有属性可以发现所有属性比公有属性多出了一个键为 stack_trace_symbol 的属性，这个属性就是 **specialObject** 的一个私有属性。下一节将介绍与私有属性有关的一个 v8 引擎的逻辑错误。

3. 微小的逻辑错误

在介绍这个逻辑错误之前，先了解下 Object.assign 这个方法,根据 ECMAScript/262 的解释[1]:

The assign function is used to copy the values of all of the enumerable own properties from one or more source objects to a target object

那么问题来了，私有属性是 v8 引擎内部使用的属性，其他 JavaScript 引擎可能根本就不存在私有属性，私有属性是否应该是可枚举的，私有属性应不应该在赋值时被拷贝，ECMAScript 根本就没有做规定。我猜 v8 的开发人员在实现 Object.assign 时也没有很周密的考虑过这个问题。私有属性是供 v8 引擎内部使用的属性，一个对象的私有属性不应该能被赋给另一个对象，否则会导致私有属性的值被用户 JavaScript 修改。v8 是一个高性能的 JavaScript 引擎，为了追求高性能，很多函数的实现都有两个通道，一个快速通道和一个慢速通道，当一定的条件被满足时，v8 引擎会采用快速通道以提高性能，因为使用快速通道出现漏洞的情况有不少先例，如 CVE-2015-6764[2]、 CVE-2016-1646 都是因为走快速通道而出现的问题。同样，在实现 Object.assign 时，v8 也对其实现了快速通道,如下代码所示[3]：

```
MUST_USE_RESULT Maybe<bool> FastAssign(Handle<JSReceiver> to,
                                       Handle<Object> next_source) {
 //detect if fast path can be used
 ……
  Handle<DescriptorArray> descriptors(map->instance_descriptors(), isolate);
  int length = map->NumberOfOwnDescriptors();
  bool stable = true;
  for (int i = 0; i < length; i++) {
    Handle<Name> next_key(descriptors->GetKey(i), isolate); ---->hasn't filtered the keys that
are private symbols and enumerable
    Handle<Object> prop_value;
    //copy all properties from next_source to target
    ……
  }
  return Just(true);
```

```
}
```

代码片段 2：逻辑错误

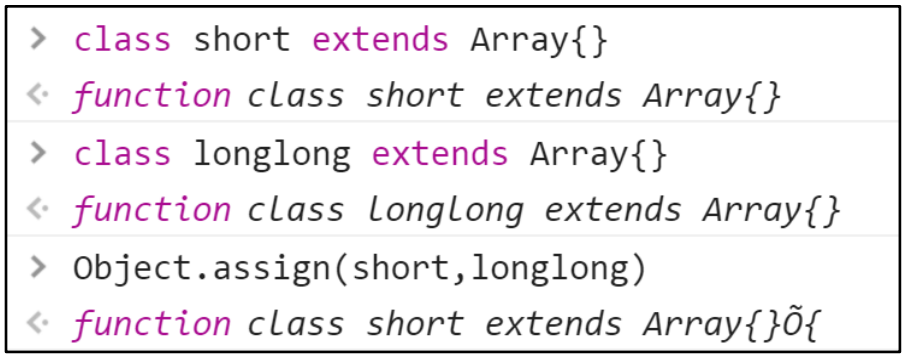
在 Object.assign 的快速通道的实现中，首先会判断当前赋值是否满足走快速通道的条件，如果不满足，则直接返回失败走慢速通道，如果满足则会简单的将源对象的所有属性都赋给目标对象，并没有过滤那些键是私有符号并且具有可枚举特性的属性。如果目标对象也具有相同的私有属性，则会造成私有属性重新赋值。这就是本文要讨论的逻辑错误。Google 对这个错误的修复很简单［4］，给对象增加任何属性时，如果此属性是私有属性，则给此属性增加不可枚举特性。现在蝴蝶已经找到了，那它如何扇动翅膀可以实现远程任意代码执行呢，我们从第一扇开始，将逻辑错误转化为越界读漏洞。

4. 从逻辑错误到越界读

现在我们有了将对象的可枚举私有属性重赋值的能力，为了利用这种能力，我遍历了 v8 中所有的私有符号[5]，尝试给以这些私有符号为键的私有属性重新赋值，希望能能搅乱 v8 引擎的内部执行流程，令人失望的是我并没有多大收获，不过有两个私有符号引起了我的注意，它们是 class_start_position_symbol 和 class_end_position_symbol，从这两个符号的前缀我们猜测这两个私有符号可能与 JavaScript 中的 class 有关。于是我们定义了一个 class 来观察它的所有属性。

```
d8> class c extends Array{}
d8> %DebugPrint(c)
DebugPrint: 0x30590e99: [Function]
....
 - properties = {
   #length: 0x453cef99 <AccessorInfo> (accessor constant)
   #name: 0x453cefd1 <AccessorInfo> (accessor constant)
   #prototype: 0x453cf009 <AccessorInfo> (accessor constant)
   0x453854c9 <Symbol: home_object_symbol>: 0x30590ebd <a c with map 0x53d098d5>
(data field at offset 0)
   0x45385335 <Symbol: class_start_position_symbol>: 0 (data field at offset 1)  ------->
   0x453852fd <Symbol: class_end_position_symbol>: 23 (data field at offset 2)  ------->
 }
```

果不其然，新定义的 class 中确实存在这两个私有属性。从键的名字和值可以猜测这两个属性决定了 class 的定义在源码中的起止位置。现在我们可以通过给这两个属性重新赋值来实现越界读。



上图是在 Chrome 54.0.2840.99 的 console 中的运行输出结果，最后一行等同于 short.toString()的结果，我们可以看到，最后一行的最后两个字符不正常，它们是发生了越界读的结果。可以通过 substr 方法得到越界字符串的一个子串，使这个子串完全是未初始化内存或者部分是初始化内存部分是未初始化化内存都是可行的。

5. 从越界读到越界写

在检查了所有其他私有符号后，并没有发现其他有意义的私有属性重赋值可被利用，现在我们唯一的收获是有了一个越界读漏洞，那么一个越界读漏洞可以转换为越界写吗？听起来匪夷所思，但在一定条件下是可以的。第四节的最后我们得到了一个可越界读的字符串 short.toString()，而在 JavaScript 中，字符串是不可变的，每次对它的修改（如 append）都会返回一个新的字符串，那么如何使用这个可越界读的字符串实现越界写呢？首先我们需要了解这样一个事实，因为这是一个越界的字符串，而在程序执行时垃圾回收，内存分配操作是随机，所以越界部分的字符是不确定的，多次访问同一个越界的字符串返回的字符串内容可能是不一样的，这就间接使得字符串是可变的。然后需要了解 JavaScript 中的一组函数，escape 和 unescape，他们分别实现对字符串的编码和解码。unescape 在 v8 中的内部实现如下[6]：

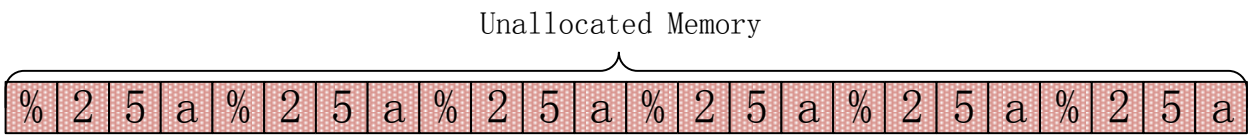
```
template <typename Char>                                                    <step 1>
MaybeHandle<String> UnescapeSlow(Isolate* isolate, Handle<String> string,
                                 int start_index) {  ------------------------------>assume the argument string is an oob string constructed above.
  bool one_byte = true;
  int length = string->length();
  int unescaped_length = 0;
  {
    DisallowHeapAllocation no_allocation;
    Vector<const Char> vector = string->GetCharVector<Char>();
    for (int i = start_index; i < length; unescaped_length++) {
      int step;
      if (UnescapeChar(vector, i, length, &step) >
          String::kMaxOneByteCharCode) {
        one_byte = false;
      }
      i += step;  ----------------------->the unescaped length of the destination string is calculated here.
    }
  }
  DCHECK(start_index < length);
  Handle<String> first_part =
      isolate->factory()->NewProperSubString(string, 0, start_index);
```

```
  int dest_position = 0;
  Handle<String> second_part;
DCHECK(unescaped_length <= String::kMaxLength);

if (one_byte) {                                                                    <step 2>
    Handle<SeqOneByteString> dest = isolate->factory()
                                    ->NewRawOneByteString(unescaped_length)
                                    .ToHandleChecked();   ------------>allocate a string using the length calculated above, but the function
NewRawOneByteString may modify the oob string, which cause the required length is larger than unescaped_length;

DisallowHeapAllocation no_allocation;                                              <step 3>
    Vector<const Char> vector = string->GetCharVector<Char>();
    for (int i = start_index; i < length; dest_position++) {
      int step;
      dest->SeqOneByteStringSet(dest_position,
                                UnescapeChar(vector, i, length, &step));       ------------------>oob write will occur here
      i += step;
    }
    second_part = dest;
  } else {
    //...
  }
  return isolate->factory()->NewConsString(first_part, second_part);
}
```

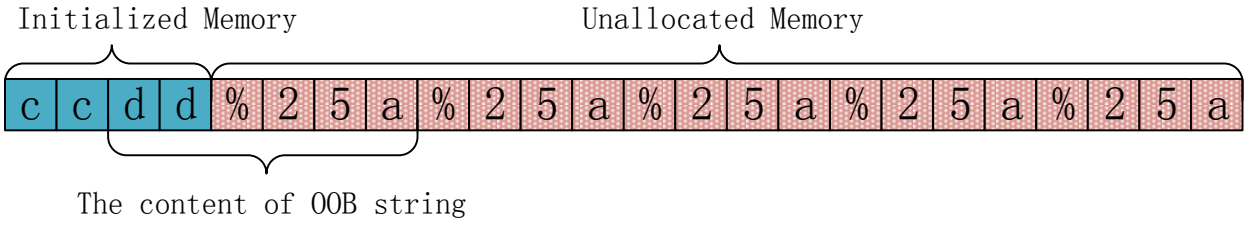
代码片段 3：unescape 的内部实现

unescape 的 v8 内部实现可以分为三步，假设输入参数 string 是我们前面构造的越界字符串，第一步是计算这个字符串解码后需要的存储空间大小；第二步分配空间用来存储解码后的字符串；第三步进行真正的解码操作。第一步和第三步都扫描了整个输入串，但因为输入是一个越界串，第一步和第三步扫描的字符串的内容可能不一样，从而导致第一步计算出的长度并不是第三步所需要的长度，从而使第三步解码时发生越界写。需要注意的是，这个函数的实现并没有问题，根本原因是输入的字符串是一个越界串，这个越界串的内容是不确定的。我们举例来说明越界到底是如何发生的。因为 v8 新分配的对象都位于 New Space[7]，New Space 采用的垃圾回收算法是 Cheney's algorithm[8],所以 New Space 中对象的分配是顺序分配的。假设我们已经将 New Space 喷满字符串”%25a”,越界写的执行流程示意如下：

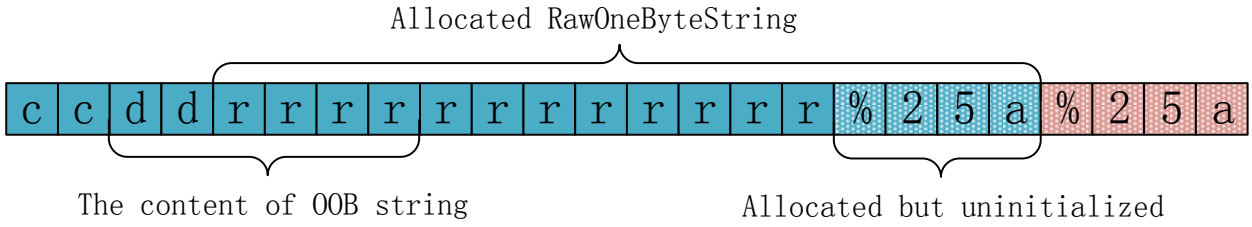
a)下图为初始内存状态，全是未分配内存，内容为喷满的”%25a”字符串；



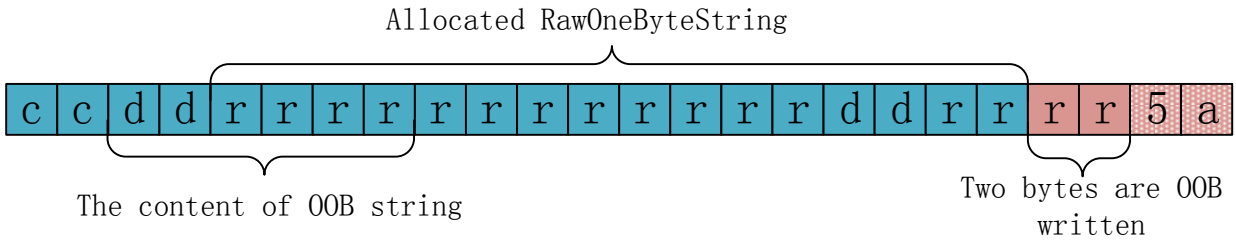
b)下图为在创建了越界串之后，在执行 unescape 之前的内存状态，假设创建的越界串的内容为“dd%25a”,其中”dd”位于已初始化的内存空间中,”%25a”位于未分配的内存中；



c)下图为在执行了代码片段 3 的第二步后的内存状态，r 代表随机值。分配的 RawOneByteString 为 16 字节，包括 12 字节的头部和 4 字节的解码后的字符（因为第一次访问越界字符串时内容为”dd%25a”,所以计算的解码后的字符串应该是“dd%a”,为四个字节）



d)下图为执行完代码片段 3 的第三步后的内存状态，也就是完成 unescape 后的内存状态，因为在执行完第二步后越界字符串的内容已经变为”ddrrrr”,r 是随机值，一般不会是字符’%’,所以解码后的字符串仍然是”ddrrrr”,导致两个字符的越界写。



6. 从越界写到任意地址读写

从越界读到越界写是整个利用过程中最巧妙的一环，但从越界写到任意地址读写却是最难的一步。一个越界写漏洞要能被利用必须有三个必要条件，长度可控，写的源内容可控，被覆盖的目的内容可控。对这个漏洞而言，前两个条件很容易满足，但要满足第三个条件颇费周折。

从上一节的最后一个图中可以看到，越界写覆盖的两个字节是未分配的内存。因为 v8 中在 New Space 中分配对象是顺序分配的，而在代码片段 3 的第二步和第三步之间没有分配任何对

象，所有 RawOneByteString 后总是未分配的内存空间，改写未分配的内存数据没有任何意义。那么如何使 RawOneByteString 对象后的内容是有意义的数据就成了从越界写到任意地址写的关键。

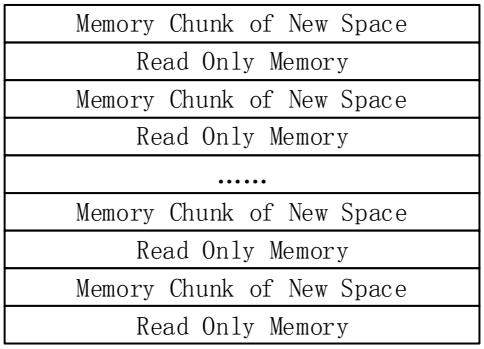
首先想到的是能不能控制在分配 RawOneByteString 时触发一次 GC，使得分配的 RawOneByteString 被重新拷贝，从而使得它之后的内存是已分配的其它对象，经过深入分析后发现此路不通，因为一个新分配的对象的第一次 GC 拷贝只是在两个半空间(from space 和 to space)之间移动，拷贝后还是在 New Space 内部，拷贝后 RawOneByteString 之后的内存依然是未分配的内存数据。

第二种思路是越界写时写过 New Space 的边界，改写非 New Space 内存的数据。这需要跟在 New Space 后的内存区间是被映射的内存并且是可写的。New Space 的内存范围是不连续的，它的基本块的大小为 1MB，最大可以达到 16MB，所以越界写时可以选择写过任意一个基本块的边界。我们需要通过地址空间布局将我们需要被覆盖的内容被映射到一个 New Space 基本块之后。将一个 Large Space[7]的基本块映射到 NewSpace 基本块之后是一个比较好的选择，这样可以能覆盖 Large Space 中的堆对象。不过这里有个障碍，我们应该记得，当第一个参数为 NULL 时，mmap 映射内存是总是返回 mm->mmap_base 到 TASK_SIZE 之间能够满足映射大小范围的最高地址，也就是说一般多次 mmap 时返回的地址应该是连续的，这样的特性很有利于操纵内存空间布局，但很不幸的是，chrome 在分配堆的基本块时，第一个参数给的是随机值，如下代码所示[9]：

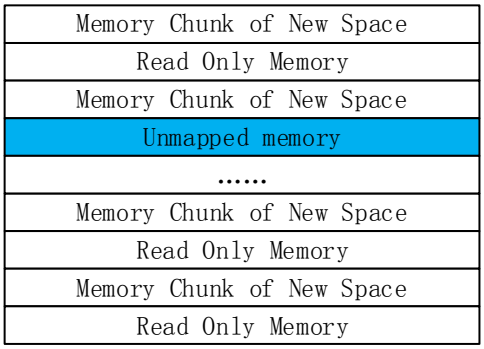
```
VirtualMemory::VirtualMemory(size_t size, size_t alignment)
    : address_(NULL), size_(0) {
  DCHECK((alignment % OS::AllocateAlignment()) == 0);
  size_t request_size = RoundUp(size + alignment,
                                static_cast<intptr_t>(OS::AllocateAlignment()));
  void* reservation = mmap(OS::GetRandomMmapAddr(),  ---->the first arguments isn't NULL
                           request_size,
                           PROT_NONE,
                           MAP_PRIVATE | MAP_ANONYMOUS | MAP_NORESERVE,
                           kMmapFd,
                           kMmapFdOffset);
  //...
}
```

这使得 New Space 和 Large Space 分配的基本块总是随机的，Large Space 的基本块刚好位于 New Space 之后后几率很小。我们采取了两个技巧来保证 Large Space 基本块刚好分配在 New Space 基本块之后。

第一个技巧是使用 web worker 绕开不能进行地址空间布局的情形；New Space 起始保留地址是 1MB，为一个基本块，随着分配的对象的增加，最大可以增加到 16MB，这 16 个基本块是不连续的，但一旦增加到 16MB，它的地址范围就已经确定了，不能再修改，如果此时 New Space 的内存布局如下图所示：



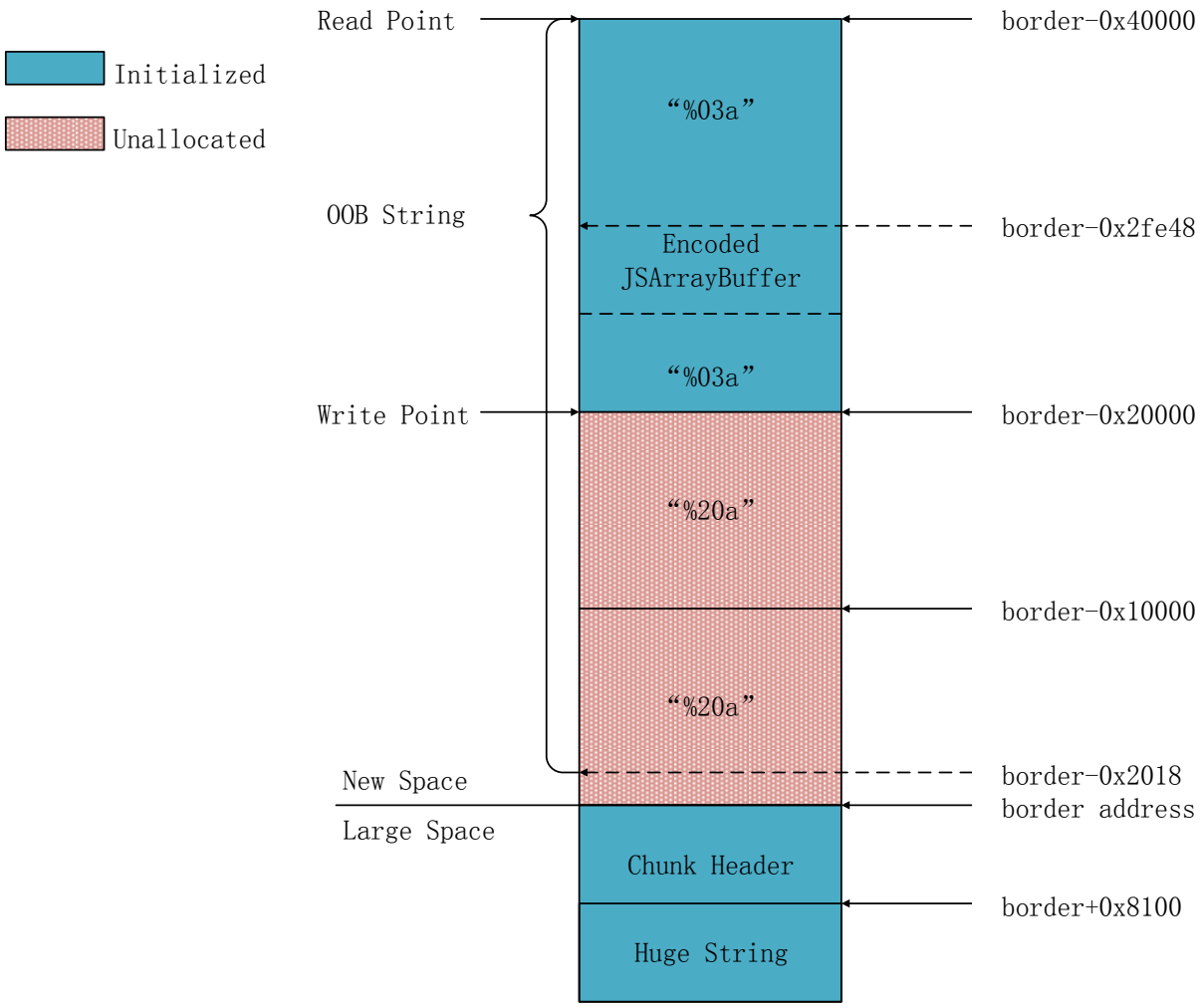
即每一个 New Space 的基本块后都映射了一个只读的内存空间，这样无论怎样进行地址空间布局都不能在 New Space 之后映射 Large Space，我们采用了 web worker 来避免产生这种状态，因为 web worker 是一个单独的 JS 实例，每一个 web worker 的 New Space 的地址空间都不一样，如果当前 web worker 处于上图所示状态，我们将结束此次利用，重新启动一个新的 webworker 来进行利用，期望新的 web worker 内存布局处于以下状态，至少有一个 New Space 基本块之后是没有映射的内存地址空间：



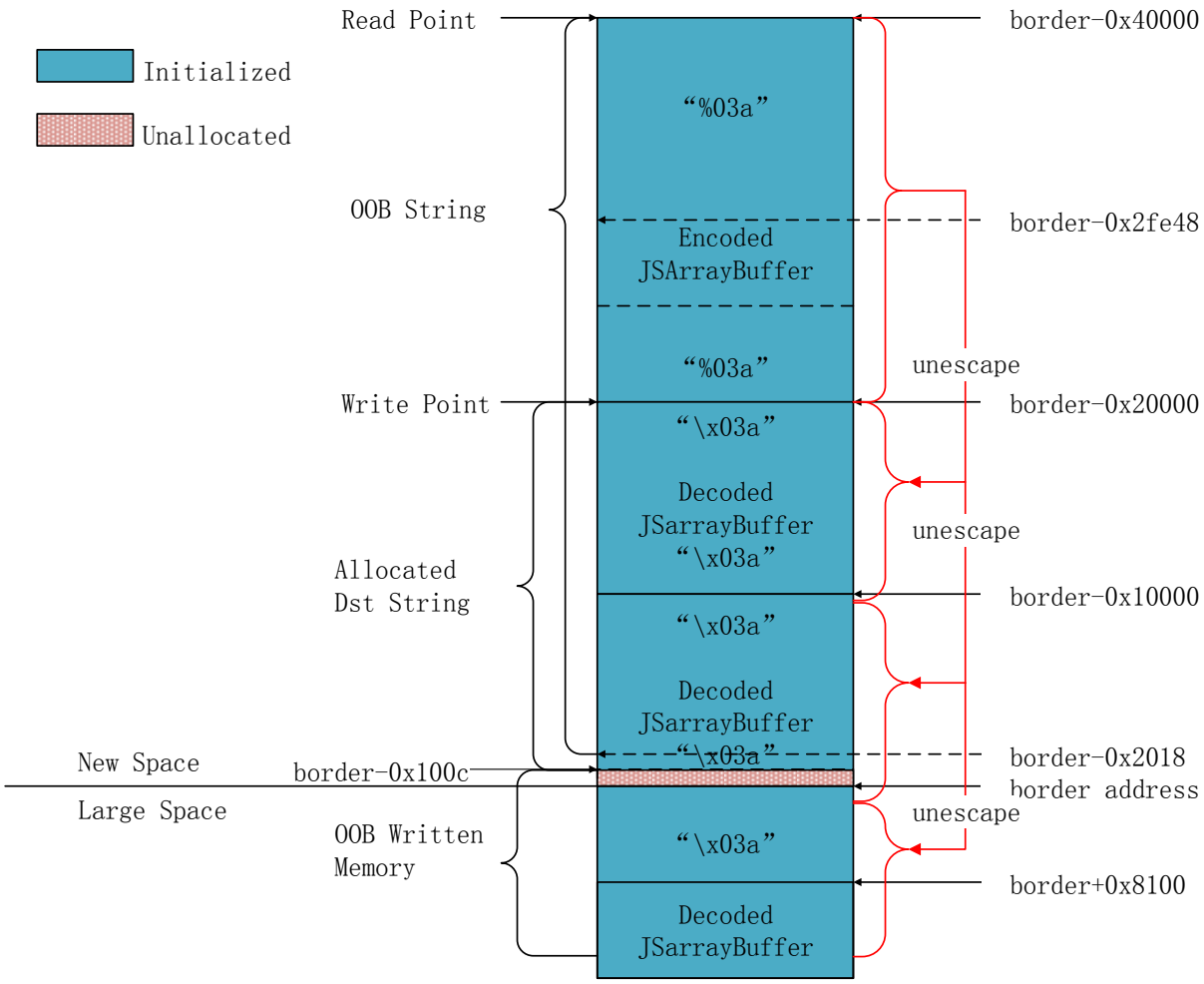
现在使用第二个技巧，我将它称为暴力风水，这与堆喷射不太一样，堆喷是指将地址空间喷满，但 chrome 对喷射有一定的限制，它对分配的 v8 对象和 dom 对象的总内存大小有限制，往往是还没将地址空间喷满，chrome 就已经自动崩溃退出了。暴力风水的方法如下：先得到 16 个 New Space 基本块的地址，然后触发映射一个 Large Space 基本块，我们通过分配一个超长字符串来分配一个 Large Space 基本块；判断此 Large Space 基本块是否位于某一 New Space 基本块之后，若不是，则释放此 Large Space 基本块，重新分配一个 Large Space 基本块进行判断，直到条件满足，记住满足条件的 Large Space 基本块之上的 New Space 基本块的地址，在此 New Space 基本块中触发越界写，覆盖紧随其后的 Large Space 基本块。

当在 v8 中分配一个特别大（大于 kMaxRegularHeapObjectSize==507136）的 JS 对象时，这个对象会分配在 Large Space 中，在 Large Space 基本块中，分配的 v8 对象离基本块的首地址的偏移是 0x8100，基本块的前 0x8100 个字节是基本块的头，要实现任意地址读写，我们只需要将 Large Space 中的超长字符串对象修改成 JSArrayBuffer 对象即可，但在改写前需要保存基本块的头，在改写后恢复，这样才能保证改写只修改了对象，没有破坏基本块的元数据。要精确的

覆盖 Large Space 基本块中的超长字符串，根据 unescape 的解码规则有个较复杂的数学计算，下图是执行 unescap 前的内存示意图：



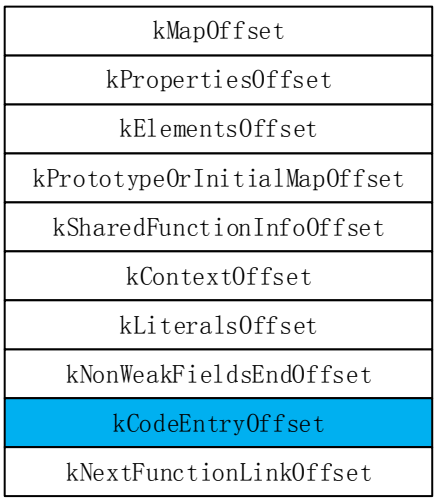
假设 Large Space 基本块的起始地址为 border address，border address 之上是 New Space,之下是 Large Space, 需要被覆盖的超长字符串对象位于 border+0x8100 位置，我们构造一个越界串，它的起始地址为 border-0x40000,结束地址为 border-0x2018,其中 border-0x40000 到 border-0x20000 范围是已分配并已初始化的内存，存储了编码后的 JSArrayBuffer 对象和辅助填充数据”%03a”, border-0x20000 到 border-0x2018 是未分配内存，存取的数据为堆喷后的残留数据”%20a”, 整个越界串的内容都是以”%xxy”的形式存在，y 不是字符%,整个越界串的长度为(0x40000-0x2018)，所以 unescape 代码片段 3 中第一步计算出的目的字符串的长度为(0x40000-0x2018)/2,起始地址为 border-0x20000，执行完 unescape 后的内存示意图如下：



在执行完代码片段 3 第二步后，Write Point 指向 border-0x20000+0xc，因为 NewRawOneByteString 创建的对象的起始地址为 border-0x20000,对象头为 12 个字节。 我们将代码片段 3 的第三步人为地再分成三步，第一步，解码从 border-0x40000 到 border-0x20000 的内容，因为此区间的内容为”%xxy”形式，所以解码后长度会减半，解码后写的地址范围为 border-0x20000+0xc 到 border-0x10000+0xc，解码后的 JSArrayBuffer 位于此区间的 border-0x17f18；第二步，解码从 border-0x20000 到 border-0x10000 的内容，因为此时此区间不含%号，所以解码只是简单拷贝，解码后长度不变，解码后写的地址范围为 border-0x10000+0xc 到 border+0xc，解码后的 JSArrayBuffer 位于此区间的 border-0x7f0c，第三步，解码从 border-0x10000 到 border-0x2018(越界串的边界)的内容，这步解码还是简单拷贝，解码后写的地址范围为 border+0xc 到 border+0xdfe8，解码后的 JSArrayBuffer 正好位于 border+0x8100，覆盖了在 Large Space 中的超长字符串对象。在 JavaScript 空间引用此字符串其实是引用了一个恶意构造的 JSArrayBuffer 对象，通过这个 JSArrayBuffer 对象可以很容易实现任意地址读写,就不再赘述。

7. 任意地址读写到任意代码执行

现在已经有了任意地址读写的能力，要将这种能力转为任意代码执行非常容易，这一步也是所有步骤中最容易的一步。Chrome 中的 JIT 代码所在的页具有 rwx 属性，我们只需找到这样的页，覆盖 JIT 代码即可以执行 ShellCode。找到 JIT 代码也很容易，下图是 JSFunction 对象的内存布局，其中 kCodeEnryOffset 所指的地址既是 JSFucntion 对象的 JIT 代码的地址。



8. 总结

这篇文章从一个微小的逻辑漏洞出发，详细介绍了如何克服重重阻碍，利用这个漏洞实现稳定的任意代码执行。文中所述的将一个越界读漏洞转换为越界写漏洞的思路，应该也可以被一些其他的信息泄露漏洞所使用，希望对大家有所帮助。

对于漏洞的具体利用，此文中还有很多细节没有提及，真正的利用流程远比文中所述复杂，感兴趣的可以去看这个漏洞的详细利用[10]。

引用

[1]https://www.ecma-international.org/ecma-262/7.0/index.html#sec-object.assign
[2]https://github.com/secmob/cansecwest2016/blob/master/Pwn%20a%20Nexus%20device%20with%20a%20single%20vulnerability.pdf
[3]https://chromium.googlesource.com/v8/v8/+/chromium/2840/src/builtins/builtins-object.cc#65
[4]https://codereview.chromium.org/2499593002/diff/1/src/lookup.cc
[5]https://chromium.googlesource.com/v8/v8/+/chromium/2840/src/heap-symbols.h#160
[6]https://chromium.googlesource.com/v8/v8/+/chromium/2840/src/uri.cc#333
[7]http://jayconrod.com/posts/55/a-tour-of-v8-garbage-collection
[8]https://en.wikipedia.org/wiki/Cheney's_algorithm
[9]https://chromium.googlesource.com/v8/v8/+/chromium/2840//src/base/platform/platform-linux.cc#227
[10]https://github.com/secmob/pwnfest2016