Butterfly Effect and Program Mistake

Exploit an "Unexploitable" Chrome Bug

----by @oldfresher @360 Alpha Team

1. Introduction

Does the flap of a butterfly's wings in Brazil set off a tornado in Texas? I don't know. But I do know a negligible tiny logical bug in v8 engine can lead to remote code execution in Chrome. In PwnFest contest 2016, I exploited a logical mistake (CVE-2016-9651) in V8 to allow remote code execution. This logical mistake was very small and it appeared unexploitable at first glance. But by the combination of several unusual exploitation tricks, I got a stable exploit at last. The journey of exploiting this vulnerability tells me: Never give up easily on "unexploitable" bugs.

This paper is organized as follows: Section 2 describes the "invisible" private property of object in V8 engine ;Section 3 tells us where is the logical mistake; Section 4 introduces how the logical mistake can be turned into out-of-bound read bug; Section 5 introduces one method how to turn an out-of-bound read bug to an out-of-bound write bug, which is the most valuable part of the whole exploitation; Section 6 is the hardest step of all links, it details a method named "brute force" Feng Shui to Shape the whole memory space and how to turn OOB write into arbitrary memory read and write; Section 7 describes converting the ability of arbitrary memory read and write to arbitrary code execution; Section 8 concludes the paper.

2. Invisible Private Property

In JavaScript, every object is an associative array. An associative array is simply a set of key value pairs. These key value pairs are also named as object's properties. A key of a property can be a string or a symbol, just as follows:

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

code snippet 1: Object properties

The code snippet above defines an object **normalObject,** and then appends two properties to the object. The properties which can be read and modified by normal JavaScript are named public property. The keys of all public properties of an object can be obtained by two methods of Object constructor. The following code snippet gets the keys of all public properties of **normalObject.**

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

Execution result：`ownPublicKeys==["string", Symbol(d)]`

In V8 JavaScript engine, in addition to public property, some special JavaScript objects have some special properties, these properties can only be accessed by the engine, they are not visible to normal JavaScript. This type of property is named as private property. In the V8 engine, symbols are divided into two types too, public symbol and private symbol. Public symbol can be created and manipulated by normal JavaScript but private symbol can only be created by the engine for internal use. Private property often uses private symbol as key. Because normal JavaScript in V8 cannot touch private symbol directly, it can't access private property through private symbol either. Since private property is invisible, How it can be observed? D8 is the Shell of V8 engine. All the properties of an object can be shown in D8 by calling the runtime function DebugPrint . For example, we can see all the properties of **normalObject** by the following command**:**

```
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)
 }
```

As d8 output shown above, normalObject has only two public properties, no private property. Now we view the properties of a special object which is error object.

```
d8> var specialObject = new Error("test");
d8> var ownNames = Object.getOwnPropertyNames(specialObject);
d8> var ownSymbols = Object.getOwnPropertySymbols(specialObject);
d8> var ownKeys = ownNames.concat(ownSymbols)
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)
 }
```

By comparing public properties and all the properties of **specialObject,** we can find there is a property named stack_trace_symbol which is among all properties but not among public properties. This property is a private property. The next section describes a logical error related to private property in v8 engine.

3. The Tiny Logical Bug

Before introducing the logical Bug, We have to understand the method Object.assign firstly. According to the interpretation of ECMAScript/262[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

There is a question. Private property is the property used by v8 internally only. Other Javascript engines maybe have no private property. Should private property be enumerated? Should private property be copied on assignment? ECMAScript hasn't defined them. I guess v8 developers had not considered this issue carefully when implementing Object.assign. Private property is for internal use only. It's should never been assigned to another object by normal JavaScript. Otherwise it will cause the value of private property to be modified which is dangerous. V8 is a high-performance JavaScript engine, in pursuit of high performance, the implementations of many functions have two paths, a fast path and a slow path. When certain conditions are met, v8 engine will take the fast path to improve performance. There were many vulnerabilities related with fast path before, such as CVE-2015-6764 [2], CVE-2016-1646 and so on. Similarly, When implementing Object.assign, v8 also implement a fast path, as shown in the code below:

```
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);
}
```

Code snippet 2 : The logical mistake

In the function FastAssign, the arguments are checked to determine if certain conditions are met, if not satisfied, the function simply return false to go slow path, If satisfied, all of the properties of the source objects will simply be assigned to the target object, the keys that are private symbols and enumerable haven't been filtered out. If the target object has the same private property, the private property will be modified. This is the logical error discussed in this article. The patch to this bug is very simple [4], when any property is appended to an object, if the property is private property, non-enumerable attribute is set to the property. Now the butterfly has been found, how it flap its wings can achieve remote code execution. Let's start from the first flap, convert the logic error into an out of bounds read vulnerability.

4. Turn the Logical Bug into OOB Read

Now we have the ability to re-assign value to private property. In order to take advantage of this capability, I scanned all the private symbols in v8 [5], tried to re-assign value to the related private properties, hoped to be able to disrupt the internal execution flow of v8 engine, it is disappointing that I had no big harvest except two private symbols which caught my attention, they were class_start_position_symbol and class_end_position_symbol. According to the prefix of their names, we can guess they belong to class. So let's define a class to observe its properties.

```
d8> class c {}
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)  ------->
 }
```

As expected, the class c defined above does have the two private properties. According to the key names and the values we can believe that these two properties determine the starting and ending location of the definition of the class in the source code. Now we can get the ability of OOB read by re-assigning value to these two private properties.

```
> class short {}
< function class short {}
> class longlonglong {}
< function class longlonglong {}
> Object.assign(short, longlonglong)
< function class short {}QÕm
```

Execute the above JavaScript sentence by sentence in Chrome (version 54.0.2840.99) console you'll see a strange output. The last line shown in the above figure is equivalent to short.toString (), we can see that the last three characters of the last line are not normal. They are the result of out-of-bounds read. The last line is an OOB string. A substring can be obtained from the OOB string by the method String.prototype.substr. It's easy to make an OOB substring completely or partially uninitialized memory by adjusting the arguments of the method substr.

5. Turn OOB Read to OOB Write

I checked all other private symbols, but found no other private property re-assignment can be exploited, so we have only an OOB read bug. It sounds crazy to convert an OOB read bug to an OOB write bug. But under certain conditions it is possible. We got an OOB string at the end of Section 4. However, string is immutable in JavaScript, if you want to modify a string, such as append a character to a string, a new string will be created and returned. You can't modify the out-of-bound content of the string simply by its reference. Then how to implement OOB write by the OOB string? First, we need to understand the fact that, because garbage collection and memory allocation happen at any time during program execution, the content of the OOB String is uncertain. The content may be different when accessing the OOB String multiple times, which makes string mutable indirectly. Second we need to understand a pair of functions in JavaScript, escape and unescape. The escape function encodes a string. This function makes a string portable, so it can be transmitted across any network to any computer that supports ASCII characters. The unescape function decodes an encoded string. The implementation of the unescape function in v8 is as follows:

```
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);
```

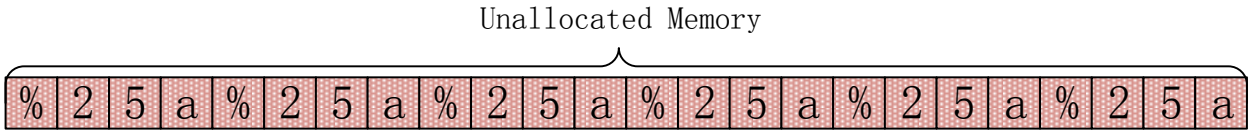
```
if (one_byte) {                                                             <step 2>
    Handle<SeqOneByteString> dest = isolate->factory()
                                ->NewRawOneByteString(unescaped_length)
                                .ToHandleChecked();   ------------>allocate a string 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);
}
```

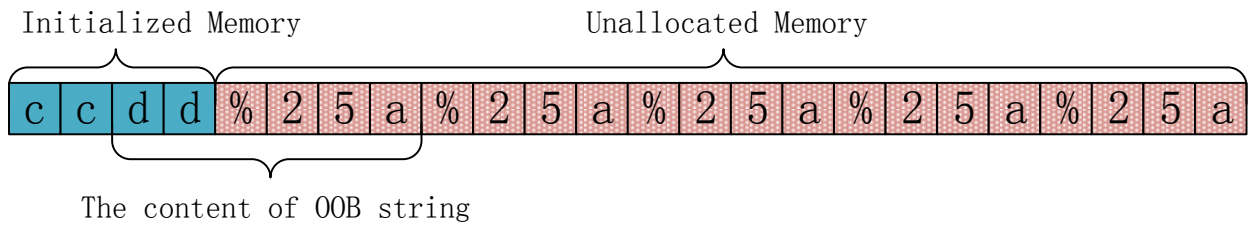
Code snippet 3: Internal implementation of unescape

The internal implementation of unescap in v8 can be divided into three steps as shown above. We assume that the input parameter string in step 1 is the OOB substring constructed at the end of Section 4. In step 1, the unescaped length of the destination string is calculated; In step 2, the destination string is allocated using the length calculated in step 1, but be careful, the NewRawOneByteString function may modify the OOB string, which may cause the required length is larger than unescaped_length; In step 3, decoded characters are written to the destination string. Step 1 and Step 2 scan the entire input string twice, but because the input string is an OOB string, the contents of the string got in step 1 and step 3 May be different, the calculated length in step 1 may be smaller than the real length step 3 required, so that OOB write will occur in step 3 while decoding. It should be noticed that the implementation of the unescape function is correct, the root cause is the input string is an OOB string which has uncertain content. To make it clear, let's illustrate how OOB write happen. Because the newly created objects are allocated in the New Space of v8 heap [7], the garbage collection algorithm in New Space is Cheney's algorithm [8], so the objects in New Space will be allocated sequentially. Suppose we have the New Space spraying full of the string "%25a", the execution flow of OOB write is as follows:

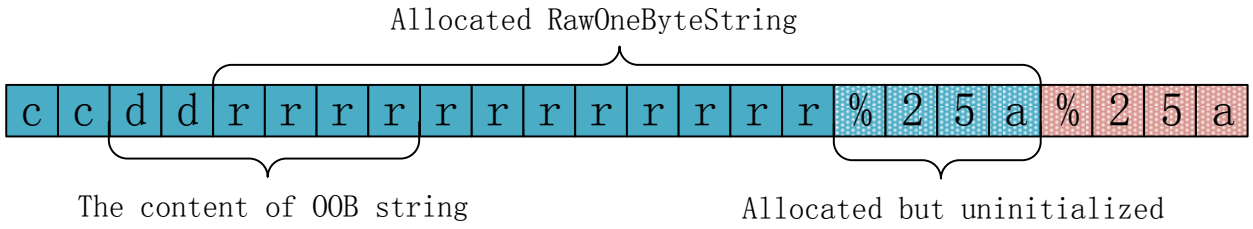
a) The following figure shows the initial memory state, all unallocated memory is sprayed to strings "%25a";



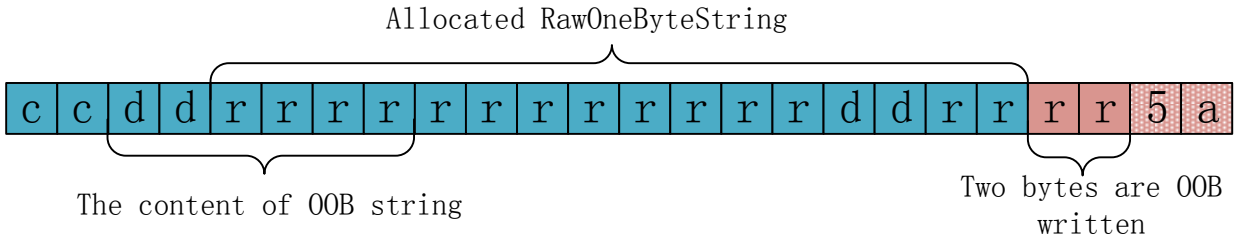
b) The following figure shows the memory state after allocating an OOB string but before executing unescape, it assumes that the content of the created OOB string is "dd% 25a", where "dd" is located in the initialized memory space, "%25a" located unallocated.



c) The following figure shows the memory status after executing the Step 3 of code snippet 3, r means random values. RawOneByteString occupies 16 bytes, including 12 bytes object head and 4 bytes to store the decoded characters (since the content of the OOB String is "dd%25a" when it's scanned firstly, so the decoded string is supposed to be "dd%a", which has 4 bytes)



d) The following figure shows the memory state after executing the step 3 of code snippet 3, because after executing the step 2 the content of the OOB string has been changed into "ddrrrr", r is a random value, generally will not be the character '%', so the decoded string is still "ddrrrr", resulting in two characters are written out of bound.



6. Turn OOB Write to Arbitrary Memory Read/Write

OOB read to OOB write described in section 5 is the most ingenious step of the whole exploitation, but turning OOB Write to Arbitrary Read/Write is the hardest step. To exploit an OOB write bug, three necessary conditions are needed: controllable length, controllable source content to be copied, controllable target content to be overwritten. To this OOB write bug, the first two conditions are easily to be met, but it's very difficult to meet the third condition.

As you can see from the last graph in the previous section, the two bytes to be OOB written are unallocated. Because the object is allocated in sequence in the New Space, and in the code snippet 3 there is no object allocated between the second step and the third step. The memory following RawOneByteString is always unallocated memory space, overwriting unallocated memory do not make any sense. So how to make the content following RawOneByteString object is meaningful data is the key to turning OOB write to arbitrary read/write.

First thought is that whether a GC could be triggered while allocating RawOneByteString, so the allocated RawOneByteString would be copied, and the memory following it would be other allocated objects. But I found it was impossible after in-depth analysis. Because a newly allocated object is copied between two semi-spaces (from space and to space) in the first GC. The relocated object is still in the

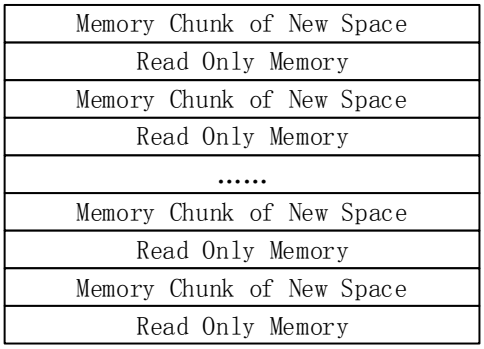
New space after copy, the memory following RawOneByteString after copying remains unallocated memory.

The second idea is to write over the boundaries of New Space when OOB write happens, rewrite the non-New Space memory data. This requires that the memory range following the New Space is mapped and is writable. New Space consists of noncontiguous memory chunks, and the size of each chunk is 1 MB. New Space has only one memory chunk when the v8 engine just start, it can be expanded to 16 at most as more objects are allocated. So any memory chunk in 16 can be chosen to trigger the OOB write in. We need to manipulate the layout of the address space to make sure the data we want to rewrite is following the selected New Space memory chunk. It's a good idea to map a memory chunk of Large Space [7] following the chosen memory chunk of New Space. So we can overwrite heap objects in Large Space. However, there is an obstacle. Recall that the behavior of mmap dictates that for every allocation, if the first parameter is NULL the chosen memory address will always be the highest address in the range of mm->mmap_base to TASK_SIZE which contains a sufficiently large contiguous unmapped region. This makes mmap an optimal allocator to use when attempting to shape the address space[11],but unfortunately, when chrome allocate memory chunk from OS, the first parameter is set to a random value, as shown in the following code snippet [9]:

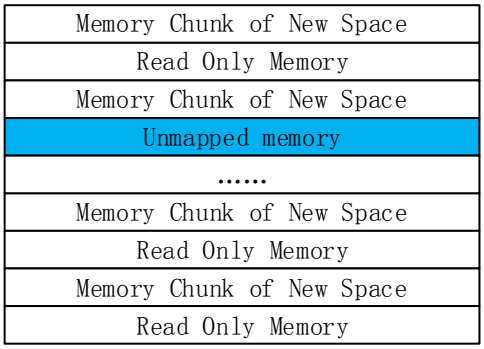
```
VirtualMemory::VirtualMemory(size_t size, size_t alignment)
    : address_(NULL), size_(0) {
  DCHECK((alignment % OS::AllocateAlignment()) == 0);
  size_t request_size = RoundUp(size + alignment,
                                static_cast<intptr_t>(OS::AllocateAlignment()));
  void* reservation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

This makes the memory chunks of New Space and Large Space are always allocated in random address. It's more difficult to shape the address space than the first parameter is NULL. We have taken two tricks to ensure one memory chunk of Large Space allocated just following the selected memory chunk of New Space.

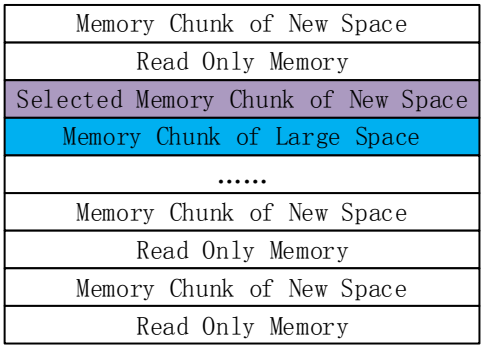
The first trick is to use the web worker to bypass the address space layout which cannot be shaped. Once New Space is expanded to 16 MB, the address of each memory chunk is determined and the chunks can't be relocated. If the New Space memory layout after expanding is as shown below, it is the situation that the address space can't be manipulated:



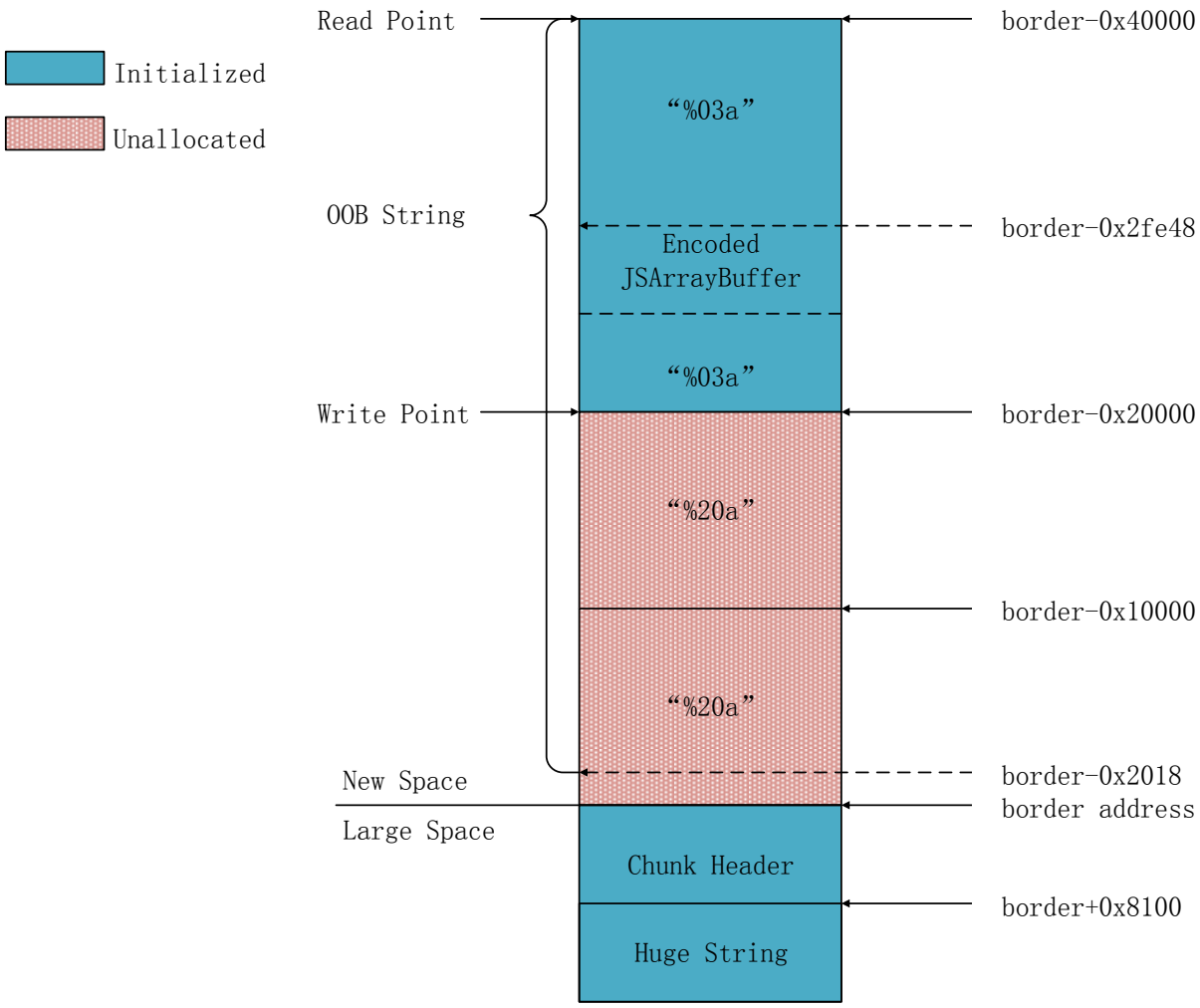
That is, following each memory chunk of the New Space is a mapped read-only memory range which can't be unmapped, so no matter how the address space is manipulated, a memory chunk of Large Space can't be laid following a memory chunk of New Space. We use web worker to bypass this situation. because web worker is a separate JS instance, each web worker's New Space's address space is not the same, if the current web worker in the state shown above, we just need to end the exploitation and create a new web worker. The memory layout of the new web worker is expected to be in the following state, with at least one memory chunk of New Space followed by an unmapped memory address space:



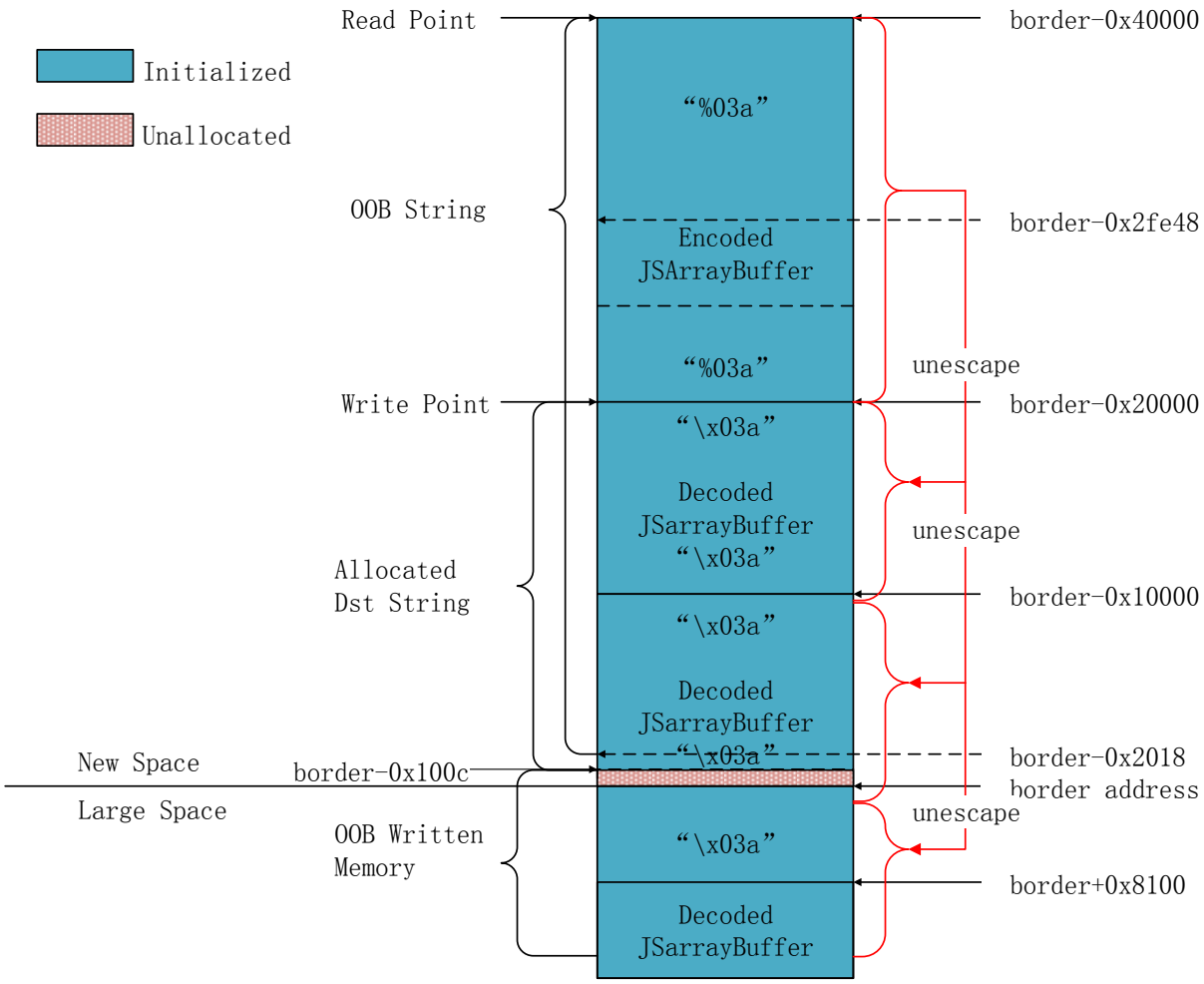
Now we need to map a memory chunk of Large Space to the position of "Unmapped memory" shown above. The first thought is occupying the "Unmapped memory" by heap spray, but it doesn't work to me, chrome restricts the total memory size allocated to v8 object and DOM object. The render process will crash before the memory space is sprayed full of Large Space's memory chunks. So we need the second trick, I named it "brute force" address space Feng Shui. We have to shape the whole memory space, not inside the heap. "Brute force" Feng Shui works as follows: first New Space is expanded to contain 16 memory chunks. The addresses of these 16 memory chunks is got by OOB read, and then trigger a memory chunk of Large Space mapped by creating a very large object; Get the address of the mapped memory chunk of Large Space by OOB read and determine if it's following one memory chunk of New Space. If not, unmap the memory chunk of Large Space by freeing the very large object, reallocate a very large object until the condition is met. Now record the New Space memory chunk followed by the just allocated Large Space memory chunk. Trigger OOB write in the selected New Space memory chunk, the following Large Space memory chunk will be overwritten. The memory layout after "brute force" Feng Shui is as follows:



When allocating a JS object such as a string that is huge (larger than kMaxRegularHeapObjectSize == 507136) in v8, the object is allocated in Large Space, the offset of the allocated v8 object from the base address of memory chunk is 0x8100, the first 0x8100 bytes is the head of the memory chunk. To implement arbitrary address read/write, we only need to modify the large object in Large Space to JSArrayBuffer object. But the head of the memory chunk have to been saved first and restored after overwriting to ensure that only the large object is rewritten, no corruption to the metadata of the memory chunk. To accurately rewrite the large object in Large Space, according to the decoding rules of the unescape function, there is a complex mathematical calculation. The following graph describe the memory status before executing the unescape function.



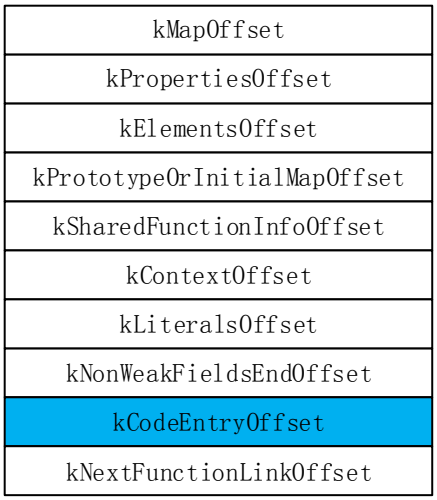
Assuming that the starting address of the memory chunk of Large Space is **border address** as shown above. New Space is above border address, Large Space is below, then the large object to be overridden is located at border+0x8100, we construct a OOB string whose start address is border-0x40000 and the end address is border-0x2018, where border-0x40000 to border-0x20000 is the allocated and initialized memory, storing the encoded JSArrayBuffer object and auxiliary fill strings "%03a", border -0x20000 to border-0x2018 is sprayed to strings "%20a" before but now is unallocated memory. the contents of the entire OOB string is in "%xxy" form, y is not the character %, the length of the whole OOB string is (0x40000-0x2018), so the length of the destination string calculated in code snippet 3 step 1 is (0x40000-0x2018) / 2, the destination string will be written from border-0x20000, after the execution of the unescape function, the memory status is as follows:



After the second step in code snippet 3, "Write Point" points to border-0x20000+0xc because the object created by NewRawOneByteString locates at border-0x20000 and the object header is 12 bytes. We artificially subdivide the third step of code snippet 3 into three steps. The first step, decoding the string content from border-0x40000 to border-0x20000, because the content of this interval is in "%xxy" form, so the length of the decoded string half , the decoded string is written to the address range of border-0x20000+0xc to border-0x10000+0xc, the decoded JSArrayBuffer is at border-0x17f18 which is in this range; The second step,decoding the string content from border-0x20000 to border-0x10000 content, because this interval does not contain the % character , so the decoding is a simple copy, the length is remain the same after decode, the decoded content is written to the address range of border-0x10000+0xc to border+0xc, JSArrayBuffer after decoding is at the address border-0x7f0c; The third step, decoding the content from border-0x10000 to border-0x2018(the end of OOB string), the decode procedure in this step is still a simple copy and the decoded content is written to the address range of border+0xc to border+0x0dfe8, after decoding the JSArrayBuffer object is copied to border+0x8100, which rewrites the large object in Large Space accurately. After rewriting, the reference to the large object in JavaScript is actually a reference to a maliciously constructed JSArrayBuffer object, it's easy to read/write arbitrary memory by the faked JSArrayBuffer object, not repeat them.

7. Turn Arbitrary Memory Read/Write to Arbitrary Code Execution

Now that you have the ability to read and write arbitrary memory, it's easy to turn that capability into arbitrary code execution, and this is the easiest step in all steps. The JIT code in Chrome has the rwx attribute on the page, and we just need to overwrite a page containing JIT code to execute the ShellCode. JIT code is also very easy to find, the following figure is the memory layout of JSFunction object, where the value at kCodeEnryOffset is the address of the JIT code.



8. Conclusion

This article starts with a small logical mistake, details how to overcome several obstacles to exploit this tiny bug to achieve stable arbitrary code execution. The thinking of convert an OOB read bug to an OOB write bug should also be used by some other information disclosure vulnerabilities.

Although this article is lengthy, there are still many smaller details it is not mentioned, the real exploitation is more complex than described, anyone who’s interested can find the source of the full exploit at my GitHub[10].

Reference

[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
[11]https://googleprojectzero.blogspot.com/2016/12/bitunmap-attacking-android-ashmem.html