Exploiting Webkit CVE-2018-4441

September 10, 2019

CVE-2018-4441 is a Webkit vulnerability that was found and reported here by lokihardt. In this post, I will explain this vulnerability and my attempt at exploiting this on the Nintendo Switch. Though, I was unable to exploit this on the console, there were a couple of other people who were interested in this. hence I am writing this post to document my analysis of this bug and what I have tried on the switch. The exploit I ended up with here is similar to @Spector's PS4 webkit exploit, as such consider this as a walkthrough of how to build an exploit for this vulnerability. This post should also help anyone new to webkit exploitation get started, everything I explain here should work with `JSC` as it is. I won't be explaining the basic webkit internals here, there are loads of amazing resources for that like Saelo's phrack paper.

I am working with the webkit revision `217062` and the Nintendo Switch OSS webkit code for `2.1.0` .

Javascript Array splice Method

For this bug, it is necessary to understand the `splice` method in Javascript. The following excerpt from W3 Schools explains this well.

```
Syntax
array.splice(index, howmany, item1, ....., itemX)

Parameter   Description
index       Required. An integer that specifies at what position to add/remove items,
            Use negative values to specify the position from the end of the array.

howmany     Optional. The number of items to be removed. If set to 0, no items will be removed

item1...X   Optional. The new item(s) to be added to the array
```

If this is still not clear, playing around with the examples on W3 Schools may help.

JavascriptCore's splice Implementation

The underlying implementation of this function is the `arrayProtoFuncSplice` function. Following is an excerpt of the relevant parts of this function, I have added comments below to explain how it works:

```
EncodedJSValue JSC_HOST_CALL arrayProtoFuncSplice(ExecState* exec)
{
    [...]
```

```
// 'index' position to add/remove items | Get clamped start index, Why? for example to translate in
unsigned begin = argumentClampedIndexFromStartOrEnd(exec, 0, length);

// 'howmany' - number of items to be removed
// By default, set to remaining number of elements from the current index
unsigned deleteCount = length - begin;

// if splice() is called with more than 1 argument, the 'howmany' argument is specified in JS
if (exec->argumentCount() > 1) {

    // get the JS passed 'howmany' value
    double deleteDouble = exec->uncheckedArgument(1).toInteger(exec);
    if (deleteDouble < 0)
        deleteCount = 0;

    // if the passed 'howmany' value is more than the remaining number of elements from index
    // ignore that and set it to the default value
    else if (deleteDouble > length - begin)
        deleteCount = length - begin;

    // otherwise all is good, use the 'howmany' value passed from JS
    else
        deleteCount = static_cast<unsigned>(deleteDouble);
}

[...]

// Equivalent to itemCount on Webkit 216891
// As seen earlier, the third argument to the splice() is a variable argument list of items
// here additionalArgs is initilialized to eitheir 0 or the count of number of items
unsigned additionalArgs = std::max<int>(exec->argumentCount() - 2, 0);

// if the number of items to be inserted is less than the number of remaining elements from the ind
// Example:
//    var arr = [ 1, 1, 1, 1, 1 ];
//    arr.splice(1, 2, '0xDEFACE');
if (additionalArgs < deleteCount) {
    // Call the underlying implementation for shift() method
    shift<JSArray::ShiftCountForSplice>(exec, thisObj, begin, deleteCount, additionalArgs, length);

    if (exec->hadException())
        return JSValue::encode(jsUndefined());
}

// if the number of items to be inserted is more than the number of remaining elements from the ind
// Example:
//    var arr = [ 1 ];
//    arr.splice(1, 2, 0xDEF4C3, 0xDEFAC3);
else if (additionalArgs > deleteCount) {
    // Call the underlying implementation for unshift() method
    unshift<JSArray::ShiftCountForSplice>(exec, thisObj, begin, deleteCount, additionalArgs, length

    if (exec->hadException())
        return JSValue::encode(jsUndefined());
```

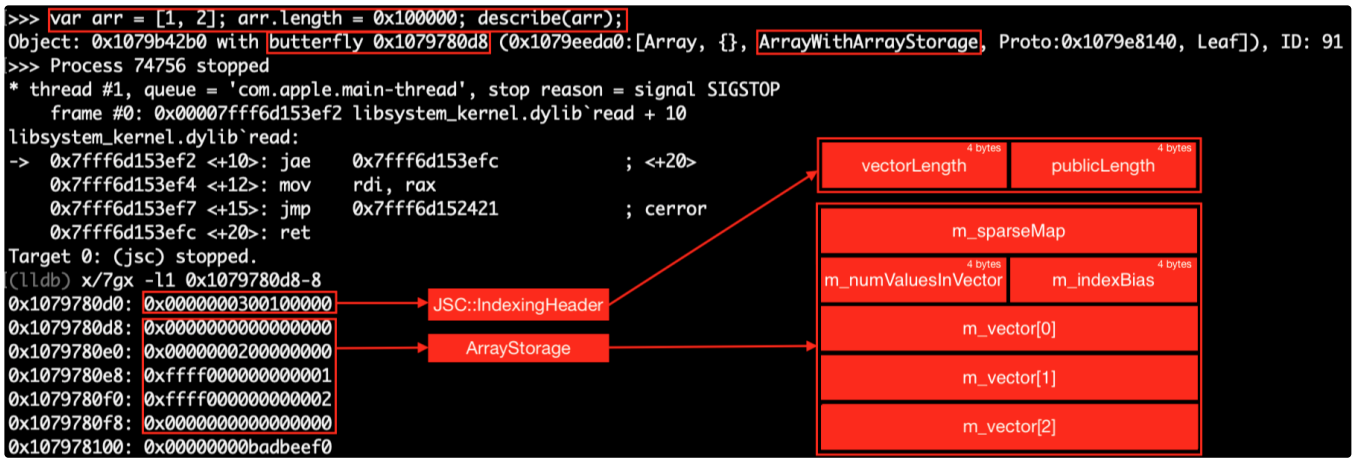
```
    }

    [...]

}
```

Understanding Arrays of type ArrayWithArrayStorage

In JavascriptCore, when the length of an array reaches `MIN_SPARSE_ARRAY_INDEX`, it transistions to the type `ArrayWithArrayStorage`. The `Butterfly` pointer of such an array points to an `ArrayStorage`. Following is the class layout for an ArrayStorage:

```
WebKit.217062 Devilx86$ ./Tools/Scripts/dump-class-layout WTF ArrayStorage
Found 1 types matching "ArrayStorage" in "/Users/Devilx86/WebKit.217062/WebKitBuild/Debug/bin/jsc"
  +0 { 24} ArrayStorage
  +0 <  8>     JSC::WriteBarrier<JSC::SparseArrayValueMap> m_sparseMap;
  +0 {  8}         JSC::WriteBarrierBase<JSC::SparseArrayValueMap>
  +0 <  8>             JSC::JSCell * m_cell;
  +8 <  4>     unsigned int m_indexBias;
 +12 <  4>     unsigned int m_numValuesInVector;
 +16 <  8>     JSC::WriteBarrier<JSC::(anonymous enum)> [1] m_vector;
Total byte size: 24
Total pad bytes: 0
```

The following debugger output shows what the butterfly pointer of such an array looks like in the memory:



Notice that in the above memory layout even though the array only has two elements, the `vectorLength` in the `indexingHeader` is `3` as such we have `m_vector[2]` in memory, which is initialised to `0`.

If we create a subarray with in one of the array element, A pointer to the `JSArray` of that subarray is stored in the `m_vector` index.

Understanding the PoC

The following is the PoC code for this vulnerability from the bug report:

```
1   function main() {
2     let arr = [1];
3
4     arr.length = 0x100000;
5     arr.splice(0, 0x11);
6
7     arr.length = 0xfffffff0;
8     arr.splice(0xfffffff0, 0, 1);
9   }
10
11  main();
```

Nintendo Switch's webkit browser is based off an old fork that does not support `ECMAScript 5` as such keywords like `let` , `const` don't work on this browser. After changing `let` to `var` , The PoC crashes the Switch's Browser.

Running the PoC in JSC also leads to a segmentation fault. In `JSC` , This crash occurs because the second `splice` method call eventually causes `JSC::JSArray::unshiftCountWithArrayStorage` method to call `clear` function on an invalid address here.

Essentially, what happens in the PoC code is:

- In `line 2` an array of type `ArrayWithDouble` is created.
- In `line 4` we set it's length property to a really large value due to which this array is then converted to an array of type `ArrayWithArrayStorage` .
- In `line 5` we call `splice` with no items i.e `additionalArgs = itemCount = 0` and `howmany = deleteCount = 0x11` , which invokes the `shift` function here. In this function, The `count` variable is based on our input to the `splice` method i.e `count = 0x11 - 0` (see here). Here, `count` is the number of elements that we actually end up adding to the array. Later, the `shiftCountWithArrayStorage` function sets the length of this array to `0x00100000 - 0x11 = 0x000fffef` , see here. After this it sets `storage->m_numValuesInVector = - count` here. Because of this, we can control `storage->m_numValuesInVector` through this first `splice` call. The initial value of `storage->m_numValuesInVector` is 1 since the array only has 1 element. So, this line now sets `storage->m_numValuesInVector` to `1 - 0x11 = 0xfffffff0` and the length of the array to `0x000fffef` . Now, `m_vector` is capable of accessing values out of bounds i.e from `0` to `0xfffffff0` elements. However note that at this point we do not have any way to manipulate or control `m_vector` to do so yet.
- In `line 7` , We set the arrays length to `0xfffffff0` which is equal to the value in `storage->m_numValuesInVector` (why? see next point).
- In `line 8` , We call the `splice` method once again, but this time with an item i.e `additionalArgs = itemCount = 1` and `howmany = deleteCount = 0` , which invokes the

`unshift` function here. This function eventually calls `unshiftCountWithArrayStorage` which has a check `storage->hasHoles()` to see if the array has any holes i.e for example if the length of an array is 5 but it only has 3 elements then the array has two "holes". This function simply checks if `storage->m_numValuesInVector` and `length` are equal or not. Hence why `line 7` is necessary and the PoC wouldn't crash without that line as the function would `return false` right at that check. Notice that here we call the `splice` method with `index = startIndex = 0xfffffff0` . Now, in the `unshiftCountWithArrayStorage` function we go without triggering any of the `memmove` function calls and invoke the `clear()` function here on `m_vector[0xfffffff0]` which does not exist , hence why the PoC causes a `segmentation fault` .

The exact flow during the second `splice` method call goes like this: `arrayProtoFuncSplice` -> `unshift` -> `unshiftCount` -> `unshiftCountForSplice` -> `unshiftCountWithAnyIndexingType` -> `unshiftCountWithArrayStorage:L964` -> `unshiftCountWithArrayStorage:L972` -> `unshiftCountWithArrayStorage:L982` -> `Segmentation fault: 11` .

Exploitation

Inside the function `unshiftCountWithArrayStorage` there are two calls to `memmove` as seen below:

```
bool JSArray::unshiftCountWithArrayStorage(ExecState* exec, unsigned startIndex, unsigned count, ArrayS
{
    [...]

    bool moveFront = !startIndex || startIndex < length / 2;
    unsigned vectorLength = storage->vectorLength();

    // Reallocates the Butterfly pointer
    if (moveFront && storage->m_indexBias >= count) {
        Butterfly* newButterfly = storage->butterfly()->unshift(structure(), count);
        storage = newButterfly->arrayStorage();
        storage->m_indexBias -= count;
        storage->setVectorLength(vectorLength + count);
        setButterflyWithoutChangingStructure(exec->vm(), newButterfly);
    }

    // if we reach this, No reallocation of the butterfly
    else if (!moveFront && vectorLength - length >= count)
        storage = storage->butterfly()->arrayStorage();

    // unshiftCountSlowCase reallocates the butterfly pointer to add an
    // uninitialized location right before the ArrayStorage
    // maybe for fast unshifts? later on
    else if (unshiftCountSlowCase(exec->vm(), moveFront, count))
        storage = arrayStorage();

    [...]

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

```
        if (startIndex) {
            if (moveFront)
                memmove(vector, vector + count, startIndex * sizeof(JSValue));
            else if (length - startIndex)
                memmove(vector + startIndex + count, vector + startIndex, (length - startIndex) * sizeof(JS
        }

        for (unsigned i = 0; i < count; i++)
            vector[i + startIndex].clear();
        return true;
    }
```

These two `memmove` function calls move elements in `m_vector` backwards. At first glance, the first `memmove` allows us to control just the `source` and the size of data to shift. furthermore, to reach the first `memmove` we need to set `moveFront` to `true` , which means it will reallocate the `Butterfly` .

The second `memmove` function on the other hand allows us to control both the `source` , `destination` and potentially also the size of data to shift. However, there are multiple issues with choosing the second `memmove` function. Firstly, to get there we will need to set `moveFront` to `false` , to do this `startIndex` will need to be a value greater than or equal to `length / 2` . This is a really large value, what happens then is in the `memove` call the destination address is set to `vector + startIndex + count` which is an address way out of the memory. Our control over `count` is limited, we cannot set `count` to a negative value (through the splice call) to adjust the `destination` address, nor can we increase the number of items in the list to overflow the value, as there is a certain limit beyond which the javascript stack runs out of memory. Hence if we try to take this path by modifying the second `splice` call to for instance: `arr.splice(-0x10, 0x0, 0x1)` , we get a `EXC_BAD_ACCESS` crash as seen below, right at the `memmove` call as the destination address we are trying to write to is huge, out of the memory and does not exist.

```
Process 82436 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = EXC_BAD_ACCESS (code=1, address=0x907b6c020)
    frame #0: 0x00007fff6d1ffed0 libsystem_platform.dylib`_platform_memmove$VARIANT$Haswell + 496
libsystem_platform.dylib`_platform_memmove$VARIANT$Haswell:
->  0x7fff6d1ffed0 <+496>: vmovups ymm0, ymmword ptr [rsi - 0x20]
    0x7fff6d1ffed5 <+501>: mov    r11, rdi
    0x7fff6d1ffed8 <+504>: sub    rdi, 0x1
    0x7fff6d1ffedc <+508>: and    rdi, -0x20
Target 0: (jsc) stopped.
(lldb) p/x $rsi
(unsigned long) $0 = 0x0000000907b6c040
```

As far as I can tell, because of the above reasons, we cannot exploit this vulnerability with the second `memmove` . I did not look into this any further and decided to go with the first `memmove` .

Now, if we choose to take the first `memmove` path, we invoke `unshiftCountSlowCase` in the `else if (unshiftCountSlowCase(exec->vm(), moveFront, count))` statement, before the shift. This function reallocates our Butterfly pointer to a new location. The following is an excerpt of this function, with just the parts relevant for this exploit, I have added comments to explain what the code does:

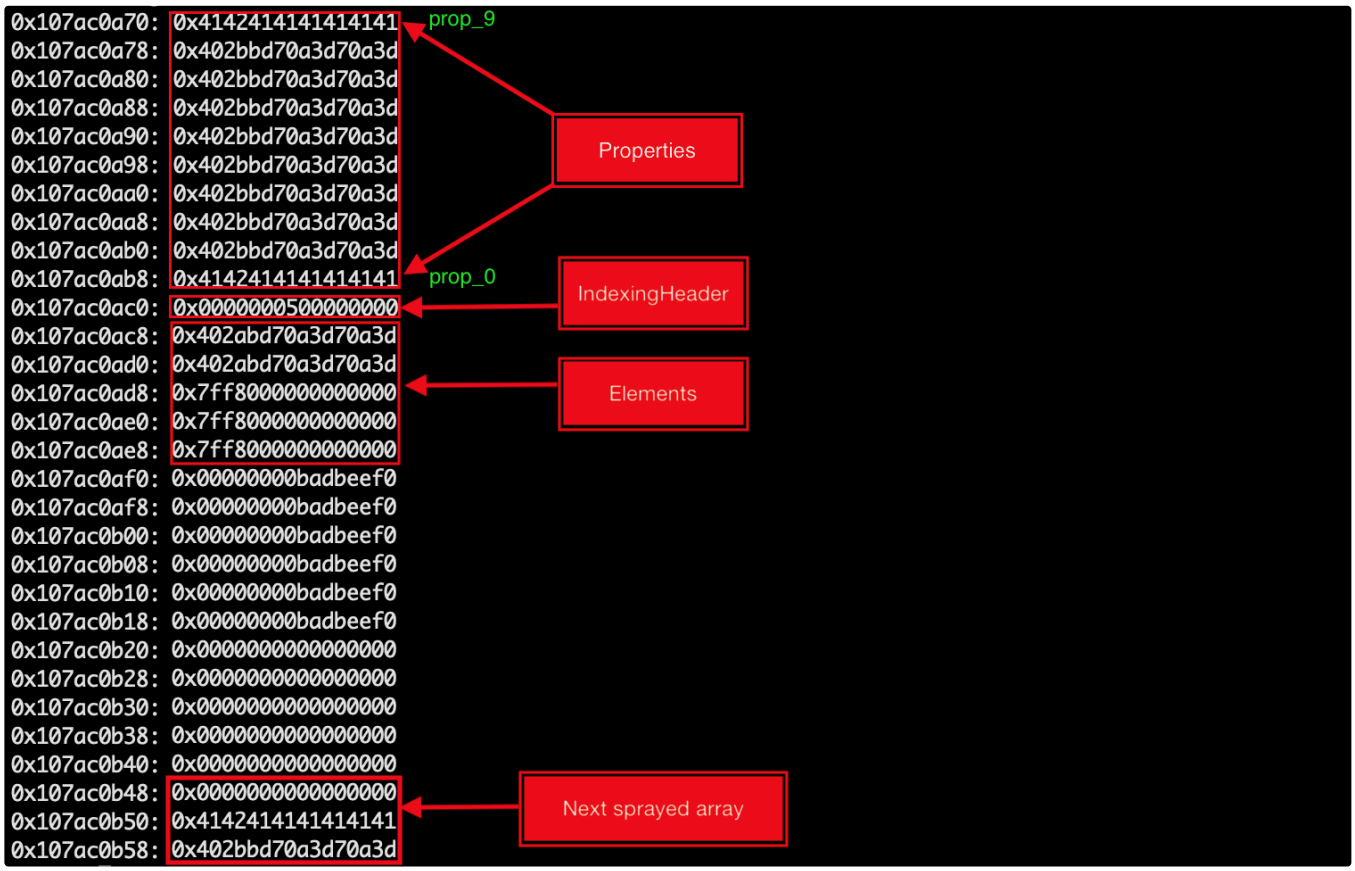
```
// This method makes room in the vector, but leaves the new space for count slots uncleared.
bool JSArray::unshiftCountSlowCase(VM& vm, bool addToFront, unsigned count)
{
    Butterfly* butterfly = storage->butterfly();

    [...]

    else {
        // ArrayStorage::sizeFor(desiredCapacity) returns 0x50 because:
        // newSize = 0x58
        size_t newSize = Butterfly::totalSize(0, propertyCapacity, true, ArrayStorage::sizeFor(desiredC

        // Allocate and get address to a region of size 0x58
        if (!vm.heap.tryAllocateStorage(this, newSize, &newAllocBase))
            return false;

        newStorageCapacity = desiredCapacity;
    }

    [...]

    // Calculate vectorLengths and IndexBias for the new butterfly
    unsigned newVectorLength = requiredVectorLength + postCapacity;
    unsigned newIndexBias = newStorageCapacity - newVectorLength;

    // Initialize a Butterfly on the newly allocated memory
    Butterfly* newButterfly = Butterfly::fromBase(newAllocBase, newIndexBias, propertyCapacity);

    if (addToFront) {
        ASSERT(count + usedVectorLength <= newVectorLength);

        // Copy the old m_vector to the new m_vector after leaving count number of spaces in the beginn
        memmove(newButterfly->arrayStorage()->m_vector + count, storage->m_vector, sizeof(JSValue) * us

        // Copy properties of the old butterfly to the new butterfly
        memmove(newButterfly->propertyStorage() - propertySize, butterfly->propertyStorage() - property
    }

    [...]
}
```

As discussed earlier, for an `ArrayWithArrayStorage` the `Butterfly` pointer points to an `ArrayStorage` and 8 bytes behind it is the `JSC::IndexingHeader` which stores the `vectorLength` and the `publicLength` of the array. Regardless of the type of an array, JavascriptCore has a `JSC::IndexingHeader` for all arrays, 8 bytes behind their `Butterfly` pointers. Our initial goal here is to spray arrays in the `CopiedSpace` so that we allocate an array right next to `m_vector` i.e within the index range of `m_vector` 's out of bounds access capabilities, then to invoke the `memmove` function so that when we move elements in `m_vector` backwards, we end up moving elements of our allocated

array to their `JSC::IndexingHeader` position, thereby corrupting the `publicLength` field of that `JSArray`.

Spray Techinique

We saw earlier that the newly allocated butterfly is of size `0x58`, Since each property in a butterfly is 8 bytes, This is exactly the size of a butterfly with 10 properties and its 8 byte IndexingHeader. Initially, I sprayed the `CopiedSpace` with a Butterfly that has 10 properties and two elements. However, this ended up corrupting one of the sprayed structure's structure ID field. The following is what the memory layout looked like during this:



Notice that the second array starts after `14 spaces`. After seeing this I sprayed arrays with `16 elements` and with butterflies of size `0x58`. This ran without any crashes, to be precise I used double elements so that we can use this array to as our `unboxed` array later on. the next step was to trigger the shift correctly to corrupt the length of some of our sprayed arrays.

[Todo: will write soon but for now see below]

To summarise, A corrupted spray element from the memory is as seen below. We can use this corrupted array to obtained out of bound read/write through the memory addresses. Later on, by setting the butterfly pointers of our boxed array and an other unboxed array to the same address (i.e sharing butterflies), we can easily cause type confusions to interprete between native values and JSValues i.e obtain `addrof` and `fakeobj` primitives. Then, using these primitives we can create a fake typed array to obtain arbitrary read/write. After this, we can use eitheir ROP or JIT pages to obtain code execution,

depending on our target.