Exploiting an integer overflow with array spreading (WebKit)

Jun 2, 2017 • By saelo (https://twitter.com/5aelo), niklasb (https://twitter.com/_niklasb)

This article is about CVE-2017-2536 (https://support.apple.com/en-us/HT207804) / ZDI-17-358 (http://www.zerodayinitiative.com/advisories/ZDI-17-358/), a classic integer overflow while computing an allocation size, leading to a heap-based buffer overflow. It was introduced in `99ed479` (https://github.com/WebKit/webkit/commit/99ed47942b1dcf935accb23b355bc8a2e93650c9), which improved the way JavaScriptCore handled ECMAScript 6 spreading operations, and discovered by saelo in February. The PoC is short enough to fit into a tweet, and we have a fully working exploit for Safari 10.1, so this is going to be fun!

The Bug

The following code is used when constructing an array through spread operations (https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/Spread_operator):

```
SLOW_PATH_DECL(slow_path_new_array_with_spread)
{
    BEGIN();
    int numItems = pc[3].u.operand;
    ASSERT(numItems >= 0);
    const BitVector& bitVector = exec->codeBlock()->unlinkedCodeBlock()->bitVector(pc[4].u

    JSValue* values = bitwise_cast<JSValue*>(&OP(2));

    // [[ 1 ]]
    unsigned arraySize = 0;
    for (int i = 0; i < numItems; i++) {
        if (bitVector.get(i)) {
            JSValue value = values[-i];
            JSFixedArray* array = jsCast<JSFixedArray*>(value);
            arraySize += array->size();
        } else
            arraySize += 1;
    }

    JSGlobalObject* globalObject = exec->lexicalGlobalObject();
    Structure* structure = globalObject->arrayStructureForIndexingTypeDuringAllocation(Arr

    JSArray* result = JSArray::tryCreateForInitializationPrivate(vm, structure, arraySize)
    CHECK_EXCEPTION();

    // [[ 2 ]]
    unsigned index = 0;
    for (int i = 0; i < numItems; i++) {
        JSValue value = values[-i];
        if (bitVector.get(i)) {
            // We are spreading.
            JSFixedArray* array = jsCast<JSFixedArray*>(value);
            for (unsigned i = 0; i < array->size(); i++) {
                RELEASE_ASSERT(array->get(i));
                result->initializeIndex(vm, index, array->get(i));
                ++index;
            }
        } else {
            // We are not spreading.
            result->initializeIndex(vm, index, value);
            ++index;
        }
    }

    RETURN(result);
}
```

At [[1]], the function computes the size of the output array, which it allocates and initializes at [[2]]. However, the size computation can overflow, causing a smaller array to be allocated. `JSObject::initializeIndex` does not perform any bounds checks as can be seen in the following piece of code:

```
/* ... */

case ALL_CONTIGUOUS_INDEXING_TYPES: {
    ASSERT(i < butterfly->publicLength());
    ASSERT(i < butterfly->vectorLength());
    butterfly->contiguous()[i].set(vm, this, v);
    break;
}

/* ... */
```

As such, a heap buffer overflow occurs. The bug can be triggered through the following script:

```
var a = new Array(0x7fffffff);
var x = [13, 37, ...a, ...a];
```

A `JSArray` of size 0 is allocated, and then 2^{32} elements get copied into it, which the browser does not like very much.

The patch (https://github.com/WebKit/webkit/commit/61dbb71d92f6a9e5a72c5f784eb5ed11495b3ff7) for this bug simply adds integer overflow checks to all affected tiers (interpreter + JITs).

Exploitation

Even though the PoC code given above uses a single array multiple times, JavaScriptCore will allocate a `JSFixedArray` for every spread operand of the array literal (in `slow_path_spread`). As such, roughly 4 billion `JSValues` will have to be allocated, taking up 32 GiB in RAM. Luckily, this isn't much of a problem due to the page compression performed by the macOS kernel (https://arstechnica.com/apple/2013/10/os-x-10-9/17/). It will, however, take roughly a minute to trigger the bug.

What is left to do now is to perform some heap feng-shui to place something interesting on the heap that we will then overflow into. We use the following heap spray to exploit the bug:

1. Allocate 100 `JSArrays` of size `0x40000` and root them (i.e. keep references). This will trigger GC multiple times and fill up holes in the heap.
2. Allocate 100 `JSArrays` of size `0x40000`, where only every second one is rooted. This triggers GC and leaves holes of size `0x40000` in the heap.
3. Allocate a larger `JSArray` and an `ArrayBuffer` of the same size. These end up directly after the spray from step 2.
4. Allocate 4 GiB of padding using `JSArrays`.

5. Trigger the bug by concatenating `JSArrays` with a combined size of 2^{32} + 0x40000 (containing the repeated byte `0x41`).

The target buffer will be allocated in the sprayed region from step 2 and the victim buffers from step 3 will be overwritten. This increases the size of the victim array to the sprayed value (`0x4141414141414141`), so that it overlaps with the victim `ArrayBuffer`. The final steps immediately yield the *fakeobj* and *addrof* primitives described in section 1.2 of the JavaScriptCore phrack paper (http://phrack.com/papers/attacking_javascript_engines.html) which can then be used to write code to a JIT page and jump to it.

In our exploit we perform step 5 in a separate web worker (https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers), so that we can launch a second stage shellcode immediately after the victim arrays are overwritten. This way we do not need to wait for the full overwrite to finish, and the heap is only left in a broken state for a very short time, so that garbage collection does not crash (which runs concurrently starting from Safari version 10.1). The full exploit can be found on our GitHub (https://github.com/phoenhex/files/blob/master/exploits/spread-overflow).

show Disqus comments