Pwn2Own 2017: UAF in JSC::CachedCall (WebKit)

May 4, 2017 • By niklasb (https://twitter.com/_niklasb), saelo (https://twitter.com/5aelo)

As a quick introduction, we are Samuel Groß (https://twitter.com/5aelo), AKA saelo, and Niklas Baumstark (https://twitter.com/_niklasb), both students at Karlsruhe Institute of Technology, and have been playing CTF together for quite some time before we decided to team up for this year's Pwn2Own. Today we are writing about a use-after-free bug in Safari 10.0.3 that could be used to get remote code execution in the browser's renderer process. This article is part of a series of write-ups we plan to do about the bugs and exploits we used to break Safari and escalate to root privileges on an up-to-date MacBook Pro.

A sad tale of two WebKit bugs

Our demo at Pwn2Own was a bit unusual in that we used a 1-day bug to get RCE inside the Safari renderer. This was due to some unfortunate timing: Around the beginning of February, saelo found a bug in the `CachedCall` class, which seemed almost impossible to exploit when we first looked at it. We then decided to try it anyways about two weeks later and ended up with a working exploit. At that point saelo tweeted the SHA-256 of a simple PoC (https://twitter.com/5aelo/status/833047999026692096), which is the file poc-cachedcall-uaf.js (https://github.com/phoenhex/files/blob/master/pocs/poc-cachedcall-uaf.js). Only 7 hours later (!) an Apple employee opened a bug report about this (https://bugs.webkit.org/show_bug.cgi?id=168567), crushing our hopes of building a full 0-day exploit chain with this bug. We don't think the developers recognized it as a security issue, since the bug is not hidden in the tracker. Later we learned that an internal fuzzer had triggered the bug, forcing them to push a fix.

With roughly one month left until the competition, we decided to concentrate on new WebKit code that would be introduced in the upcoming Safari 10.1. We succeeded in finding and exploiting a bug there, but again were a bit unfortunate because Apple decided not to release Safari 10.1 (as part of macOS 10.12.4) before the contest. We reported this second bug too and will do a write-up about it once it is fixed, since it affects the current Safari 10.1.

Overview

The WebKit bug we used at Pwn2Own is CVE-2017-2491 (https://support.apple.com/en-us/HT207600) / ZDI-17-231 (http://www.zerodayinitiative.com/advisories/ZDI-17-321/), a use-after-free of a `JSString` object in JavaScriptCore. By triggering it, we can obtain a dangling pointer to a `JSString` object in a JavaScript callback. At first, the specific scenario seems very hard to exploit, but we found a rather generic technique to still get a reliable read/write primitive out of it, although it requires a very large (~28 GiB) heap spray. This is possible even on a MacBook with 8 GB of RAM thanks to the page compression mechanism in macOS.

The following article is structured as follows:

The full, commented exploit can be found in the file cachedcall-uaf.html (https://github.com/phoenhex/files/blob/master/exploits/cachedcall-uaf.html).

The Bug

When `String.prototype.replace` (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/replace) is called with a `RegExp` object as the first argument, the following native function

(https://github.com/WebKit/webkit/blob/498268047e19b5e310afe767cf21b061a79ea780/Source/JavaScriptCore/runtime/StringPrototy
is invoked in JavaScriptCore (JSC), the JavaScript engine of WebKit:

```
static ALWAYS_INLINE EncodedJSValue replaceUsingRegExpSearch(
    VM& vm, ExecState* exec, JSString* string, JSValue searchValue, CallData& callData,
    CallType callType, String& replacementString, JSValue replaceValue)
{
    // ...

    // [[ This path is taken if the regex has the g flag set and
    //    the second argument is a JS function ]]
    if (global && callType == CallType::JS) {
        // regExp->numSubpatterns() + 1 for pattern args, + 2 for match start and string
        int argCount = regExp->numSubpatterns() + 1 + 2;
        JSFunction* func = jsCast<JSFunction*>(replaceValue);
        CachedCall cachedCall(exec, func, argCount);        // [[ 0 ]]
        RETURN_IF_EXCEPTION(scope, encodedJSValue());
        if (source.is8Bit()) {
            while (true) {
                int* ovector;
                MatchResult result = regExpConstructor->performMatch(vm, regExp, string, source, startPosition, &ove
                if (!result)
                    break;

                if (UNLIKELY(!sourceRanges.tryConstructAndAppend(lastIndex, result.start - lastIndex)))
                    OUT_OF_MEMORY(exec, scope);

                unsigned i = 0;
                for (; i < regExp->numSubpatterns() + 1; ++i) {
                    int matchStart = ovector[i * 2];
                    int matchLen = ovector[i * 2 + 1] - matchStart;

                    if (matchStart < 0)
                        cachedCall.setArgument(i, jsUndefined());
                    else
                        // [[ 1 ]]
                        cachedCall.setArgument(i, jsSubstring(&vm, source, matchStart, matchLen));
                }

                cachedCall.setArgument(i++, jsNumber(result.start));
                cachedCall.setArgument(i++, string);

                cachedCall.setThis(jsUndefined());
                JSValue jsResult = cachedCall.call();           // [[ 2 ]]
                replacements.append(jsResult.toWTFString(exec));
                RETURN_IF_EXCEPTION(scope, encodedJSValue());

                lastIndex = result.end;
                startPosition = lastIndex;

                // special case of empty match
                if (result.empty()) {
                    startPosition++;
                    if (startPosition > sourceLen)
                        break;
                }
            }
        }

    // ...
```

At [[0]], a `CachedCall` instance is created, which is later used to call the callback function. In the branch of WebKit used for Safari 10.0.3, the `CachedCall` class looked as follows
(https://github.com/WebKit/webkit/blob/498268047e19b5e310afe767cf21b061a79ea780/Source/JavaScriptCore/interpreter/CachedCal

```
class CachedCall {

    // ...

    private:
        bool m_valid;
        Interpreter* m_interpreter;
        VM& m_vm;
        VMEntryScope m_entryScope;
        ProtoCallFrame m_protoCallFrame;
        Vector<JSValue> m_arguments;
        CallFrameClosure m_closure;
};
```

As we can see, a `WTF::Vector` is used to hold the arguments, and during the algorithm above, these are the only references to the objects created by `jsSubstring`. In JavaScriptCore, all objects whose lifetime is managed by the garbage collector inherit from `JSCell`. During the marking step of the mark & sweep algorithm (https://en.wikipedia.org/wiki/Tracing_garbage_collection#Basic_algorithm), several locations are scanned for references to `JSCell`s and marked recursively. These include:

- the current call stack
- the global JavaScript execution context, including the so-called *global object*
- special buffers such as `MarkedArgumentBuffer`
- probably some others

All objects that are not reachable by traversing pointers from any of those locations are eligible to be freed during the sweeping step of the GC algorithm. This means that if the only references to a `JSCell` lie in opaque heap buffers such as a `WTF::Vector`, the garbage collector has no way of finding and marking them correctly, and they will be swept and freed if a major garbage collection cycle happens.

In the case of `String.prototype.replace`, a major GC can happen during the allocation of a new argument string at [[1]]. In that case, the previous arguments (`JSString` instances) will be collected and freed. When the call is later performed at [[2]], the callback function will receive pointers to the freed `JSCell`s as arguments. Note that the GC needs to happen *before* the callback, otherwise there will be references to the substring objects on the call stack.

The file poc-cachedcall-uaf.js (https://github.com/phoenhex/files/blob/master/pocs/poc-cachedcall-uaf.js) demonstrates the issue and works in Safari 10.0.3: At the end of the script, `i_am_free` is a pointer to a freed `JSString`. Doing anything with it other than checking its type will likely crash the browser, because its `JSCell` header has been overwritten with a free-list pointer (more on that later). The relevant source code is shown below:

```
function i_want_to_break_free() {
    var n = 0x40000;
    var m = 10;
    var regex = new RegExp("(ab)".repeat(n), "g"); // g flag to trigger the vulnerable path
    var part = "ab".repeat(n); // matches have to be at least size 2 to prevent interning
    var s = (part + "|").repeat(m);
    while (true) {
        var cnt = 0;
        var ary = [];
        s.replace(regex, function() {
            for (var i = 1; i < arguments.length-2; ++i) {
                if (typeof arguments[i] !== 'string') {
                    i_am_free = arguments[i];
                    throw "success";
                }
                ary[cnt++] = arguments[i];  // root everything to force GC
            }
            return "x";
        });
    }
}
try { i_want_to_break_free(); } catch (e) { }
console.log(typeof(i_am_free));  // will print "object"
```

The bug was fixed by replacing `Vector` with `MarkedArgumentBuffer` in the `CachedCall` class (https://github.com/WebKit/webkit/commit/7d1b3b9542d9870b8524f284e108bea56397bd3a#diff-9548589d17126311e3c48799703a4617L73).

Exploitation

Often, use-after-free bugs in browser can be exploited by turning them into a type confusion when a new object gets allocated in the freed spot. The situation is different in the context of JSC though: It operates on `JSCell` objects which contain their own type information. Hence a dangling pointer to a `JSCell` cannot be directly exploited if the freed spot is occupied by a different, newly allocated `JSCell`. Exploitation might still be possible if the freed object was keeping some other garbage collected object alive, or if the dangling pointer can be made to point to the inside of another `JSCell` via a misalignment. The former case is known from the famous Pegasus exploit (https://info.lookout.com/rs/051-ESQ-475/images/pegasus-exploits-technical-details.pdf), where a freed (but still intact) `JSArray` was used after its backing buffer had already been freed and replaced.

Both of these cases are not exhibited here: The `JSString` objects created by `jsSubstring` share their content with the original `JSString` on which `replace()` was called, and which continues to live. Furthermore, `JSString` objects are allocated inside a heap block where only other `JSCell`s of roughly the same size (24 or 32 bytes) are allocated, and all

allocations will be aligned to 32 bytes.

Our only idea for applying common techniques here is to deallocate the whole heap block containing the arena, and allocate an arena responsible for different types in its place. We never investigated this further. Instead we ended up using a different approach which is very generic and turned out to work quite well for Pwn2Own.

JSCell free-list pointer type confusion

When a `JSCell` is collected and swept, its first 8 bytes are replaced by a pointer to the next free cell in the same heap block. The other fields remain unchanged. For this reason, if we try to use the dangling pointer without allocating something over the freed `JSString` object first, we get a crash.

The first 8 bytes of a used `JSCell` contains the following fields:

```
StructureID m_structureID;           // dword
IndexingType m_indexingTypeAndMisc;  // byte
JSType m_type;                       // byte
TypeInfo::InlineTypeFlags m_flags;   // byte
CellState m_cellState;               // byte
```

At this point, the very weak heap address layout randomization of Safari comes in handy: On macOS 10.12.3, heap addresses in Safari start at around `0x110000000` to `0x120000000` and grow upward. If a pointer in this range overlaps with a `JSCell` header, the lower 32 bits of the free-list pointer will overlap with the structure ID and bits 32-39 will become the indexing type, while the three other fields are zero.

We can construct a usable `JSObject` by spraying multiple gigabytes of memory (using array buffers) such that the IndexingType becomes 8. We need to spray around 7 times 4 GiB of memory because we need an address in the range `0x800000000` to `0x8ffffffff`. Spraying this much memory is easily possible due to macOS's page compression (https://arstechnica.com/apple/2013/10/os-x-10-9/17/), but it takes about 50 seconds on the target machine used in Pwn2Own (a 2016 13.3" MacBook Pro with 16GB of RAM).

Indexing type 8 corresponds to fast contiguous storage for `JSValues` (`ContiguousShape`). Indexed accesses on this object will directly consult the butterfly of the object instead of performing a full property lookup. How butterflies work is described in section 1.2 of saelo's phrack paper (http://www.phrack.org/papers/attacking_javascript_engines.html). The butterfly pointer is the second quadword of the `JSObject`, which happens to overlap with the old string length and type flags (both 32-bit integers) of the freed `JSString` instance. In our exploit those will always produce the pointer `0x200000001`, which conveniently points inside our heap spray. The following graphic illustrates the overlap between a `JSString` and the `JSObject` which occurs after the `JSCell` header is overwritten by a heap pointer of the form `0x8xxxxxxx`:

```
Original JSString:


 JSCell fields                                          JSString fields
+-------------------------------------------------------------------------+
| dword        | byte         | byte   | byte  | byte      | dword | dword  |
| StructureID  | IndexingType | JSType | flags | CellState | flags | length |
|              |              |        |       |           |       |        |
| *            | *            | *      | *     | *         | 0x01  | 0x02   |
+-------------------------------------------------------------------------+


After header is overwritten by the pointer 0x8xxxxxxx, we get a JSObject:


 JSCell fields                                          JSObject fields
+-------------------------------------------------------------------------+
| dword        | byte         | byte   | byte  | byte      | qword          |
| StructureID  | IndexingType | JSType | flags | CellState | butterfly ptr  |
|              |              |        |       |           |                |
| xxxxxxxx     | 0x08         | 0      | 0     | 0         | 0x200000001    |
+-------------------------------------------------------------------------+
```

At this point, we have access to a fake `JSObject` with fast-path indexing, whose butterfly overlaps with an `ArrayBuffer` we control as part of our heap spray. This can be easily turned into the *fakeobj* and *addrof* primitives as described in section 4 of the phrack paper linked above, by writing to the fake `JSObject` and reading from the corresponding `ArrayBuffer`, as well as the other way around. From there we proceed as usual: We construct an arbitrary read/write primitive, overwrite the JIT code of a JavaScript function with our own shellcode, and call the function.

Some operations still access the structure of our fake `JSObject`, so we need to have a valid structure ID. The structure is retrieved through a table lookup using the structure ID as the index. The table contains pointers to `Structure` instances. Since we cannot control the structure ID itself (it overlaps with the free-list pointer), the lookup will access memory beyond the table and in our sprayed region. We thus have to create fake structure table entries in our heap spray. The free-list pointer will be 16 byte aligned, which is the granularity of the JSC allocators. Moreover, when accessing the structure instance through the structure table, the index is multiplied by 8 (pointer size). As such we only need to have a structure pointer every 128 bytes in our spray. In our exploit, we set all fake table entries to the fixed pointer `0x150000008` and create a fake structure instance at the beginning of every page in the sprayed data.

Triggering the bug

The approach from above looks straightforward at first:

1. Spray 28 GiB of memory to push heap into `0x8xxxxxxxx` region.
2. Trigger the bug.

However, the second step is not actually that easy. While the simple PoC code from above triggers almost immediately in Safari 10.0.3, this is only because the heap is very small at that point in time, so GC happens very frequently. With 28 GiB of mapped memory, it is very unlikely that an allocation still triggers GC due to heuristics in the JSC allocator.

The good thing however is that at least in the version of JSC used for Safari 10.0.3, the garbage collector is completely deterministic, even though the new concurrent garbage collector in WebKit HEAD no longer is. So we just played around until we found a combination of heap spray, regex and input string that reliably triggers the bug in the `0x8xxxxxxxx` region. In our actual exploit we only spray 14 GiB of array buffers before we start to call `String.prototype.replace` in a loop. It just so happens that this reliably leads to the situation we need, where the indexing type of a freed `JSString` ends up being overwritten with 8.

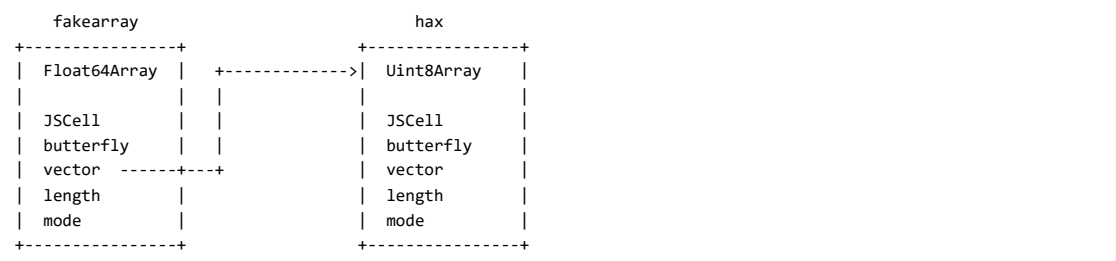
We had some ideas how to make it work even in WebKit HEAD, but then Apple fixed the bug so we stopped working on it. This means however that our exploit is very fragile with regard to the heap setup. If we change the allocation pattern of the exploit too much, the bug no longer triggers at the correct time and the exploit fails.

From fakeobj/addrof to arbitrary R/W

saelo's phrack paper (http://www.phrack.org/papers/attacking_javascript_engines.html) uses the following approach to go from being able to fake a JSC object to arbitrary read/write:

1. Spray a lot of different `Float64Array` structures so that we can guess one of those structure IDs reliably.
2. Fake a Float64Array `fakearray` with the guessed structure ID inside the inline properties of a `JSObject`. Its data pointer points to a `Uint8Array` called `hax`.
3. For a read or write, set `fakearray[2] = <target address>` and then read/write from/to `hax`.

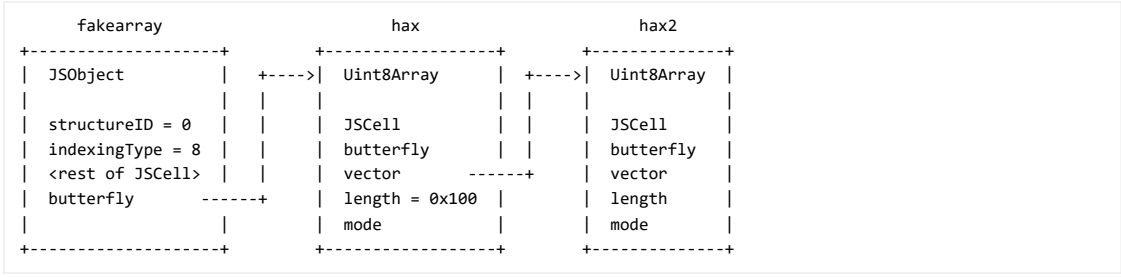
The setup looks as follows:



We tried using the same approach at first for our Pwn2Own exploit, but step 1 messed up our heap layout too much. In the interest of KISS, we just used our newly learned trick again:

1. Fake a `JSObject` called `fakearray` with indexing type 8 and structure ID 0 inside the inline properties of another object. Have its butterfly point to a `Uint8Array` called `hax`.
2. Create an extra `Uint8Array` called `hax2`.
3. Set `fakearray[2] = hax2`, thereby changing the backing buffer of `hax` to the address of `hax2`.
4. For a read/write, write the target address to `hax` at offset 16, thereby changing the backing buffer of `hax2` to the target. Then read/write to/from `hax2`.

This works because a structure with ID 0 always exists in Safari 10.0.3. The setup here looks as follows:



Surviving a completely broken heap

By allocating a `JSCell` over any of the freed `JSString`s, we effectively corrupt the free list of the heap block where the `JSString` resided. This completely breaks the allocator, so we need to make sure that we perform no allocations of size 24 or 32 in our exploit. This is easier said than done: While we can easily avoid manual allocations by not creating any objects, calling JavaScript functions or executing a loop with more than 16 iterations internally triggers certain JIT compilation tasks that perform allocations of the problematic sizes and crash JSC immediately.

It is possible to fix the broken free list and restore a somewhat reasonable heap state to work with, but for the purpose of Pwn2Own we decided to go with the route of just having a very ugly but reliable exploit code that avoids any loops, function calls/definitions or any other dangerous operations. Inside our second stage, we immediately register a signal handler for SIGSEGV, SIGBUS and SIGALRM that causes the faulting thread to sleep infinitely. This way any concurrently running threads cannot crash the process while our sandbox escape is running.

The full, commented exploit can be found in the file cachedcall-uaf.html (https://github.com/phoenhex/files/blob/master/exploits/cachedcall-uaf.html).