Improving native memory management & diagnostics in Node.js

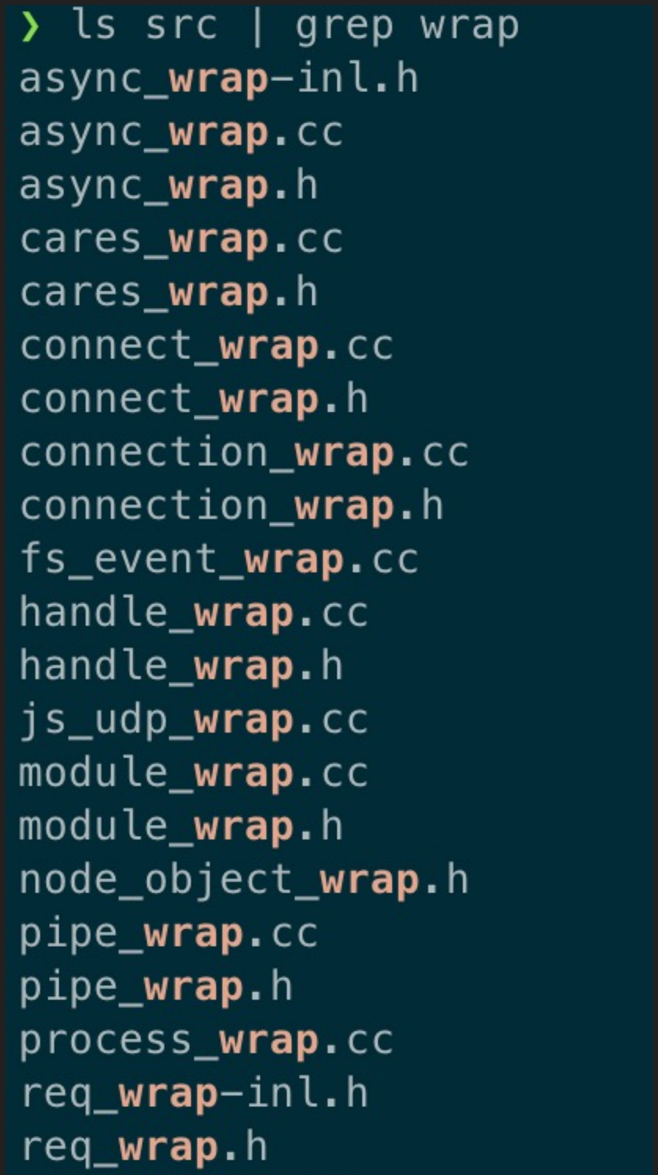
Joyee Cheung
Node.js collaboration summit, April 4, London

Agenda

- Current infrastructure for native memory management in Node.js
- Case study of dynamic import() memory issues
  - Motivation: importModuleDynamically() broken for 2 years keeping many stuck with v16.x. Many users complained about not getting a fix prioritized. Happened to find a fix/workaround. Trying to spread the knowledge out.
- A path to Oilpan (cppgc)
- Memory debugging

Current infrastructure for native memory management in Node.js

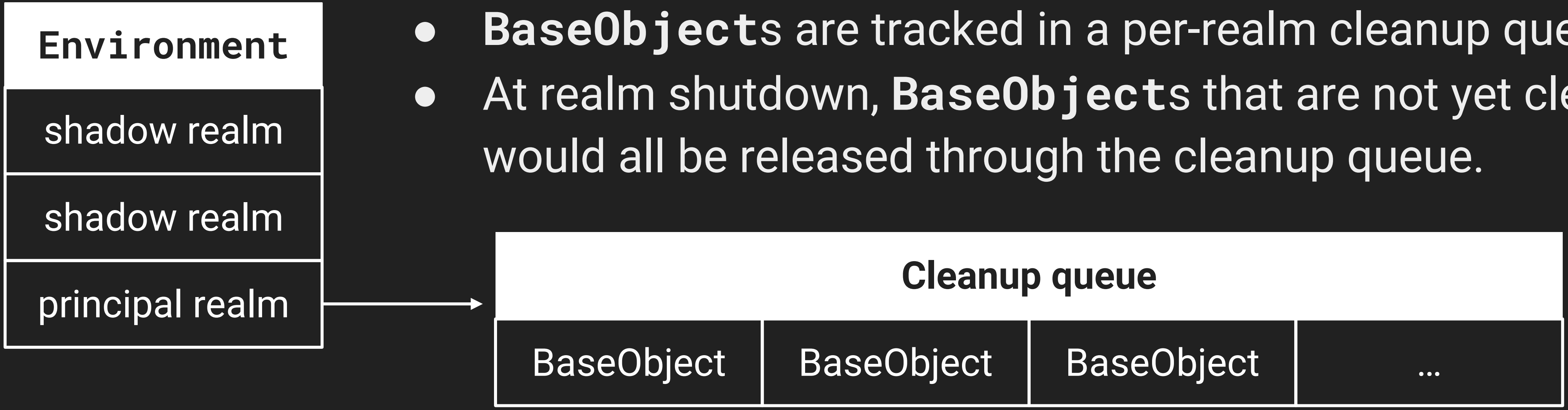
- Many internal JS land objects are created as wrappers over internal C++ objects (if you see `*Wrap` in JS or C++, it's most likely to be one)
- **`BaseObject`**: Abstraction of (almost?) all C++ object wrappers



```
❯ ls src | grep wrap
async_wrap-inl.h
async_wrap.cc
async_wrap.h
cares_wrap.cc
cares_wrap.h
connect_wrap.cc
connect_wrap.h
connection_wrap.cc
connection_wrap.h
fs_event_wrap.cc
handle_wrap.cc
handle_wrap.h
js_udp_wrap.cc
module_wrap.cc
module_wrap.h
node_object_wrap.h
pipe_wrap.cc
pipe_wrap.h
process_wrap.cc
req_wrap-inl.h
req_wrap.h
```

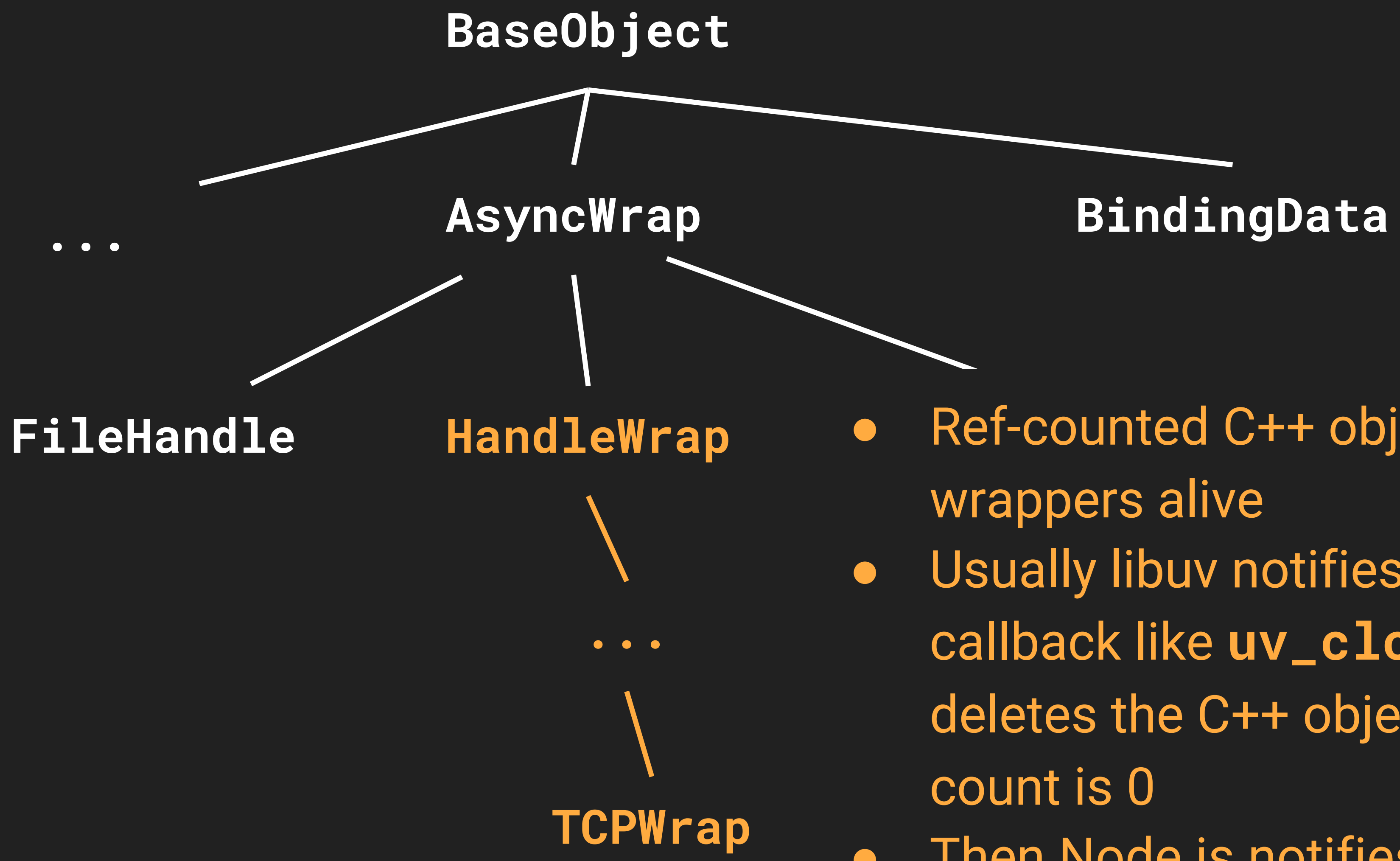
BaseObject tracking

- **BaseObjects** are tracked in a per-realm cleanup queue
- At realm shutdown, **BaseObjects** that are not yet cleared would all be released through the cleanup queue.



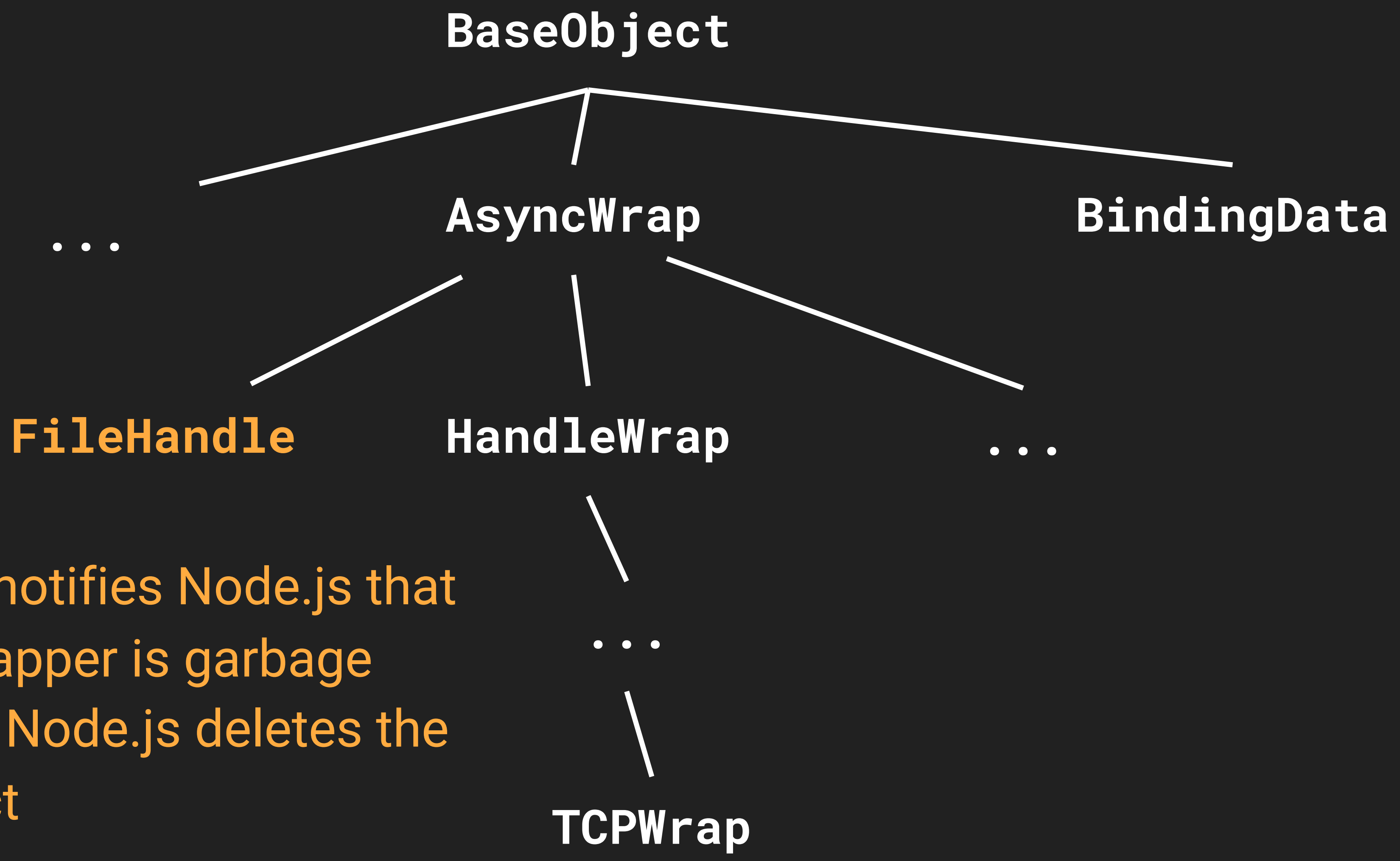
```
BaseObject::BaseObject(Realm* realm, Local<Object> object)
   : persistent_handle_(realm->isolate(), object), realm_(realm) {
 SetInternalFields(realm->isolate_data(), object, static_cast<void*>(this));
 realm->AddCleanupHook(DeleteMe, static_cast<void*>(this));
 realm->modify_base_object_count(1);
}
```

BaseObject: the base class of (almost) all wrappers



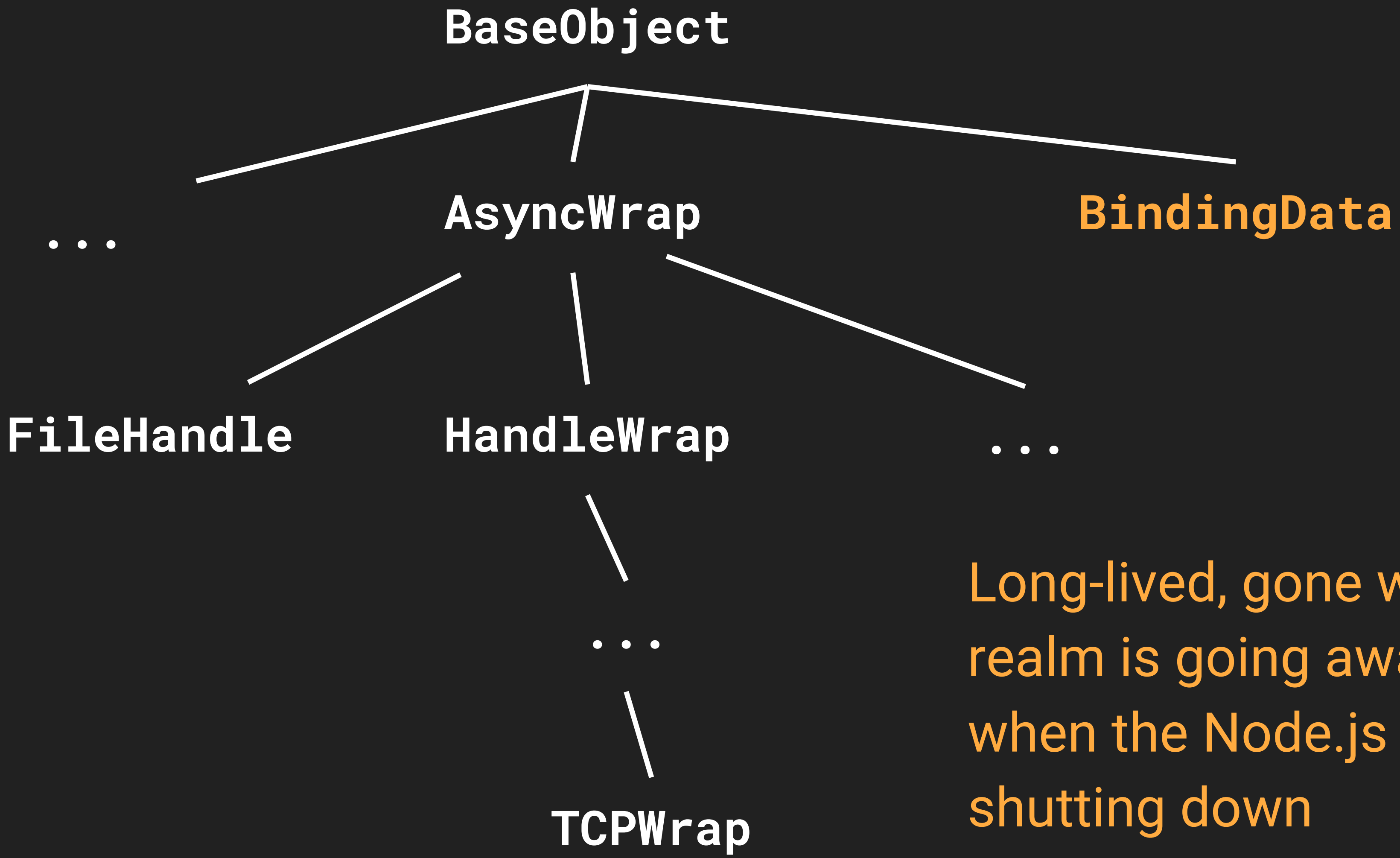
- Ref-counted C++ objects hold JS wrappers alive
- Usually libuv notifies Node.js in a callback like **`uv_close()`**, Node.js deletes the C++ object when ref count is 0
- Then Node.js notifies V8 to release the JS wrapper

BaseObject: the base class of (almost) all wrappers



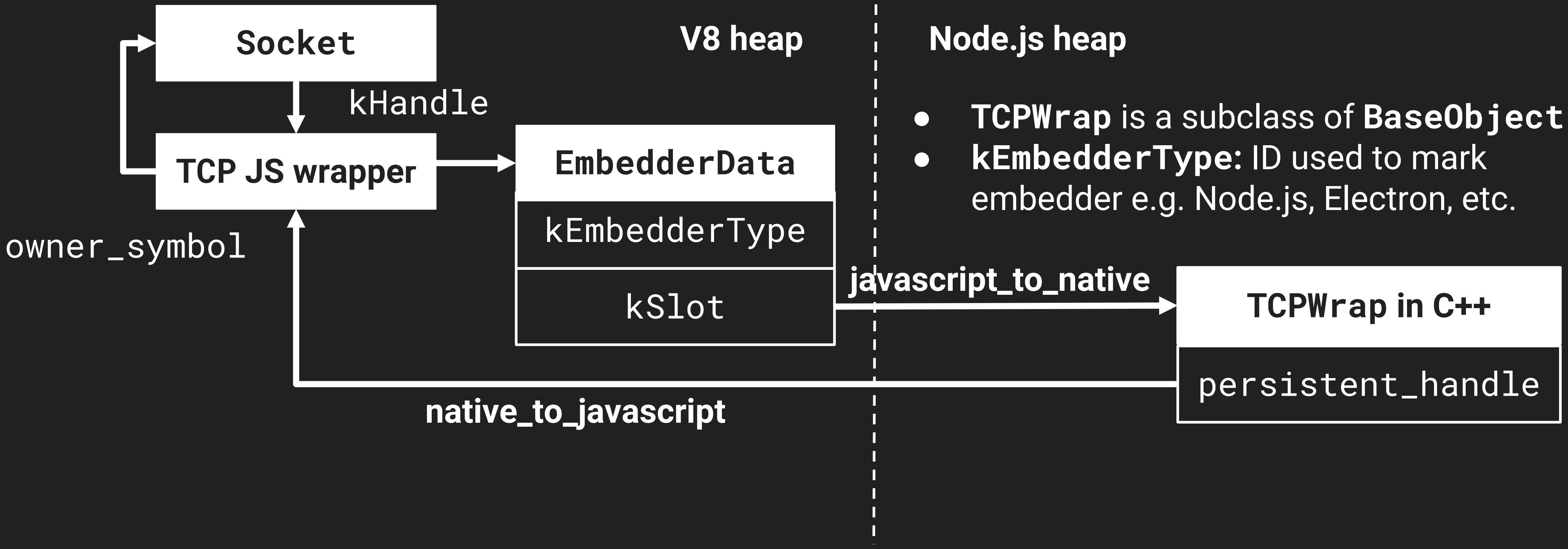
When V8 notifies Node.js that the JS wrapper is garbage collected, Node.js deletes the C++ object

BaseObject: the base class of (almost) all wrappers



Long-lived, gone when the realm is going away e.g. when the Node.js instance is shutting down

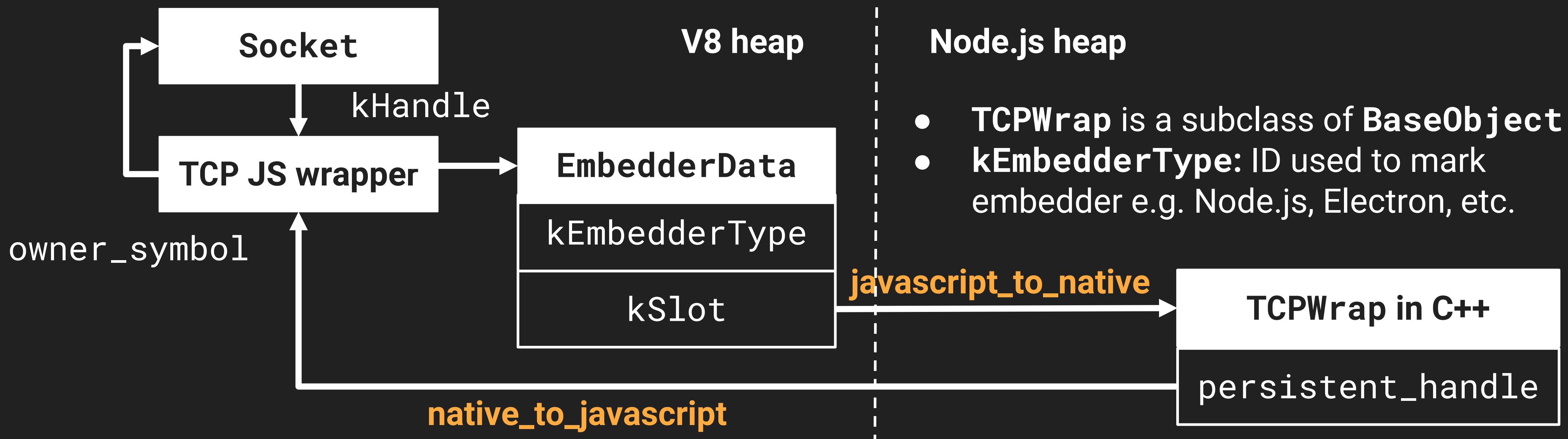
Kept alive by C++: TCPWrap & net.Socket



- **`TCPWrap`** is a subclass of **`BaseObject`**
- **`kEmbedderType`:** ID used to mark embedder e.g. Node.js, Electron, etc.

```
Socket.prototype.connect = function(...args) {
  this[kHandle] = new TCP(SOCKET);
  this[kHandle][owner_symbol] = this;
}
```

Kept alive by C++: TCPWrap & net.Socket



Socket

kHandle

TCP JS wrapper

owner_symbol

V8 heap

EmbedderData

kEmbedderType

kSlot

Node.js heap

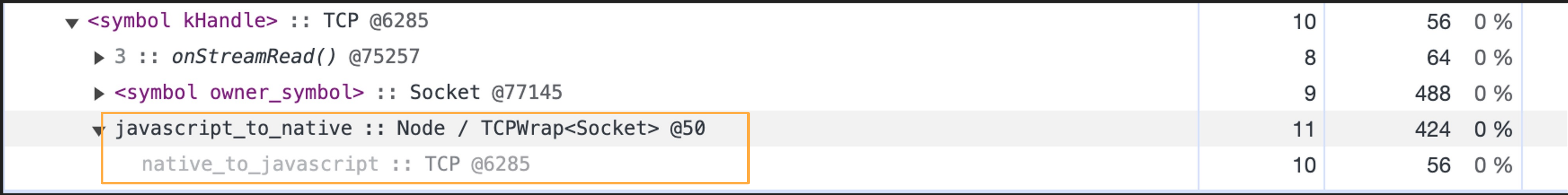
- **`TCPWrap`** is a subclass of **`BaseObject`**
- **`kEmbedderType`:** ID used to mark embedder e.g. Node.js, Electron, etc.

javascript_to_native

TCPWrap in C++

persistent_handle

native_to_javascript



```
▼ <symbol kHandle> :: TCP @6285                          10      56   0 %
   ▶ 3 :: onStreamRead() @75257                           8      64   0 %
   ▶ <symbol owner_symbol> :: Socket @77145               9     488   0 %
   ▼ javascript_to_native :: Node / TCPWrap<Socket> @50  11     424   0 %
        native_to_javascript :: TCP @6285                10      56   0 %
```

Kept alive by JS

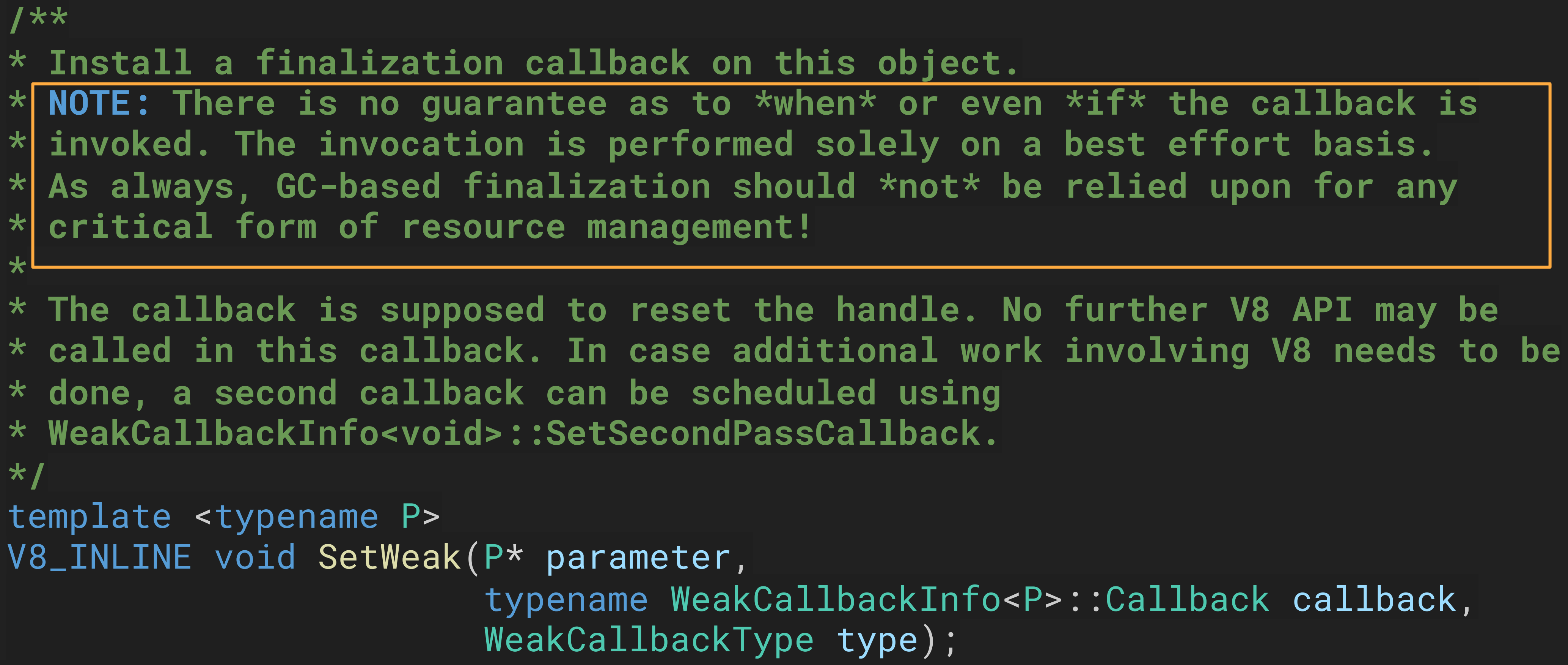
```
class BaseObject : public MemoryRetainer {
public:
 void MakeWeak() {
   persistent_handle_.SetWeak(this, [](const WeakCallbackInfo<BaseObject>& data) {
         BaseObject* obj = data.GetParameter();
         obj->persistent_handle_.Reset();
         delete obj;
       }, WeakCallbackType::kParameter);
 }
private:
 v8::Global<v8::Object> persistent_handle_;
};
```

When V8 notifies Node.js that the JS wrapper is garbage collected, Node.js delete the C++ object

If MakeWeak() isn’t called, the BaseObject is usually kept alive by the per-realm CleanupQueue

C++ -> JavaScript memory reference

- Many BaseObjects have additional references to JS values
- Callback-based memory management is also a hack (though it’s been working)

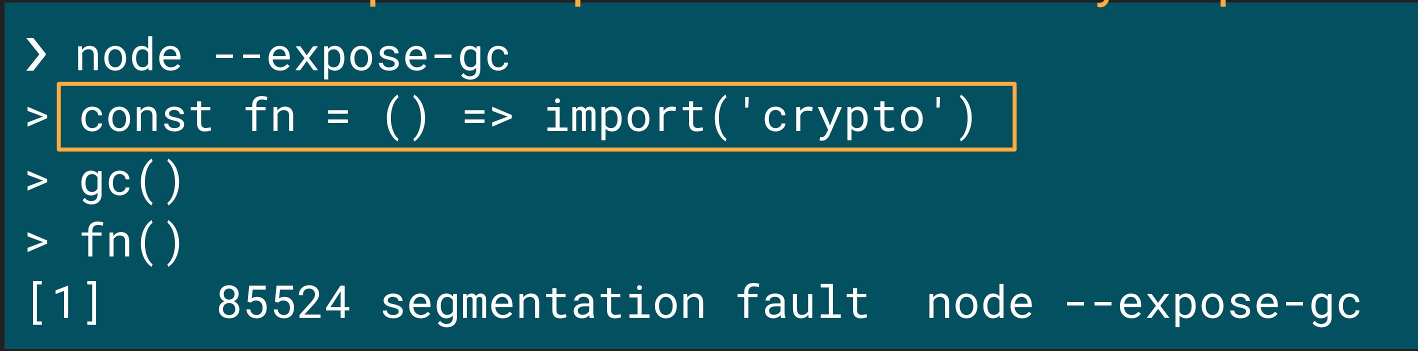


```
/**
* Install a finalization callback on this object.
* NOTE: There is no guarantee as to *when* or even *if* the callback is
* invoked. The invocation is performed solely on a best effort basis.
* As always, GC-based finalization should *not* be relied upon for any
* critical form of resource management!
*
* The callback is supposed to reset the handle. No further V8 API may be
* called in this callback. In case additional work involving V8 needs to be
* done, a second callback can be scheduled using
* WeakCallbackInfo<void>::SetSecondPassCallback.
*/
template <typename P>
V8_INLINE void SetWeak(P* parameter,
                       typename WeakCallbackInfo<P>::Callback callback,
                       WeakCallbackType type);
```

Case study of dynamic import() memory issues

```
class ContextifyScript : public BaseObject {
 public:
 ContextifyScript(Environment* env, Local<Object> object)
    : BaseObject(env, object) {
    script_.Reset(isolate, v8_script);
    MakeWeak();
 }
 private:
  v8::Global<v8::UnboundScript> script_;
}
```

REPL input compiled with a ContextifyScript



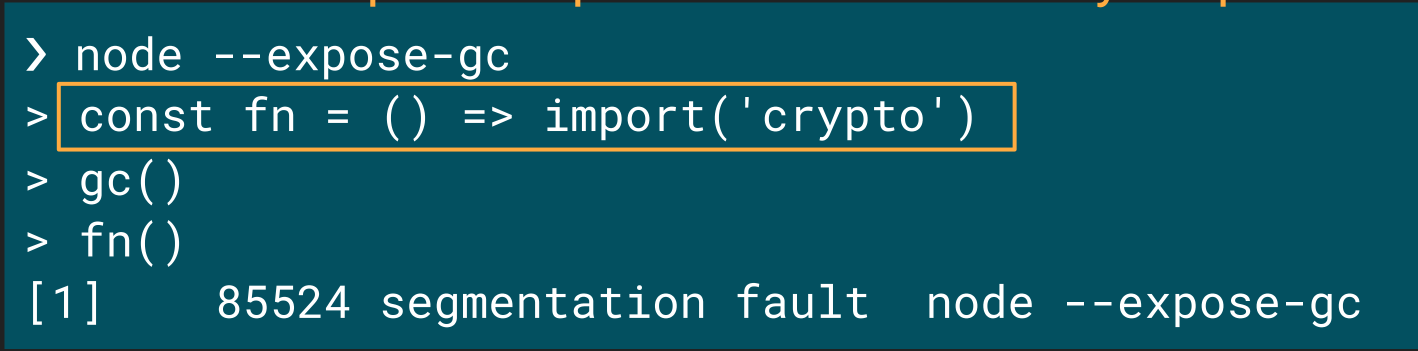
```
❯ node --expose-gc
> const fn = () => import('crypto')
> gc()
> fn()
[1]    85524 segmentation fault  node --expose-gc
```

Issue #35889

Case study of dynamic import() memory issues

```
class ContextifyScript : public BaseObject {
 public:
 ContextifyScript(Environment* env, Local<Object> object)
   : BaseObject(env, object) {
   script_.Reset(isolate, v8_script);
   MakeWeak();
 }
 private:
  v8::Global<v8::UnboundScript> script_;
}
```

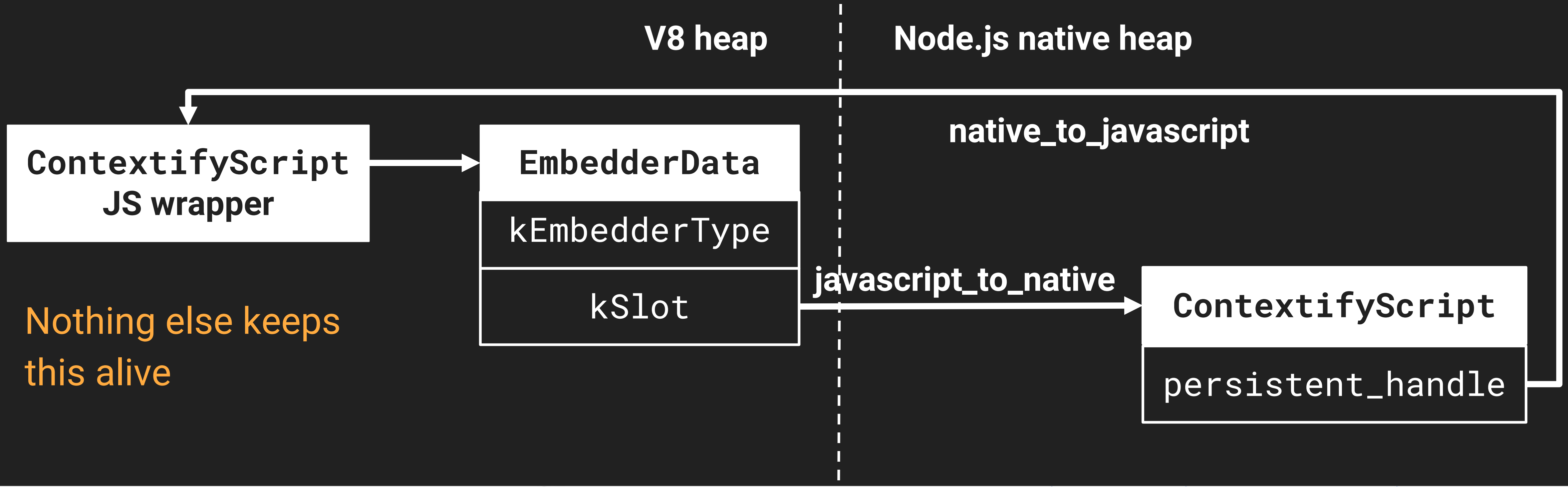
REPL input compiled with a ContextifyScript



```
❯ node --expose-gc
> const fn = () => import('crypto')
> gc()
> fn()
[1]    85524 segmentation fault  node --expose-gc
```

JS land can lose reference to the JS object, and it goes away too soon and crashes due to use-after-free

Case study of dynamic import() memory issues


V8 heap

Node.js native heap

ContextifyScript
JS wrapper

EmbedderData

kEmbedderType

kSlot

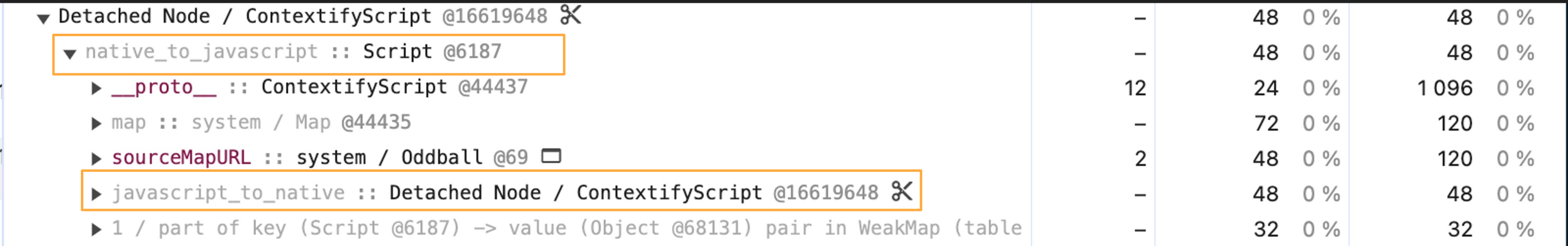
native_to_javascript

javascript_to_native

ContextifyScript

persistent_handle

Nothing else keeps this alive


```
▼ Detached Node / ContextifyScript @16619648 ✂                                   –    48  0 %     48  0 %
  ▼ native_to_javascript :: Script @6187                                        –    48  0 %     48  0 %
    ▶ __proto__ :: ContextifyScript @44437                                     12    24  0 %  1 096  0 %
    ▶ map :: system / Map @44435                                                –    72  0 %    120  0 %
    ▶ sourceMapURL :: system / Oddball @69 □                                    2    48  0 %    120  0 %
    ▶ javascript_to_native :: Detached Node / ContextifyScript @16619648 ✂       –    48  0 %     48  0 %
    ▶ 1 / part of key (Script @6187) -> value (Object @68131) pair in WeakMap (table   –    32  0 %     32  0 %
```

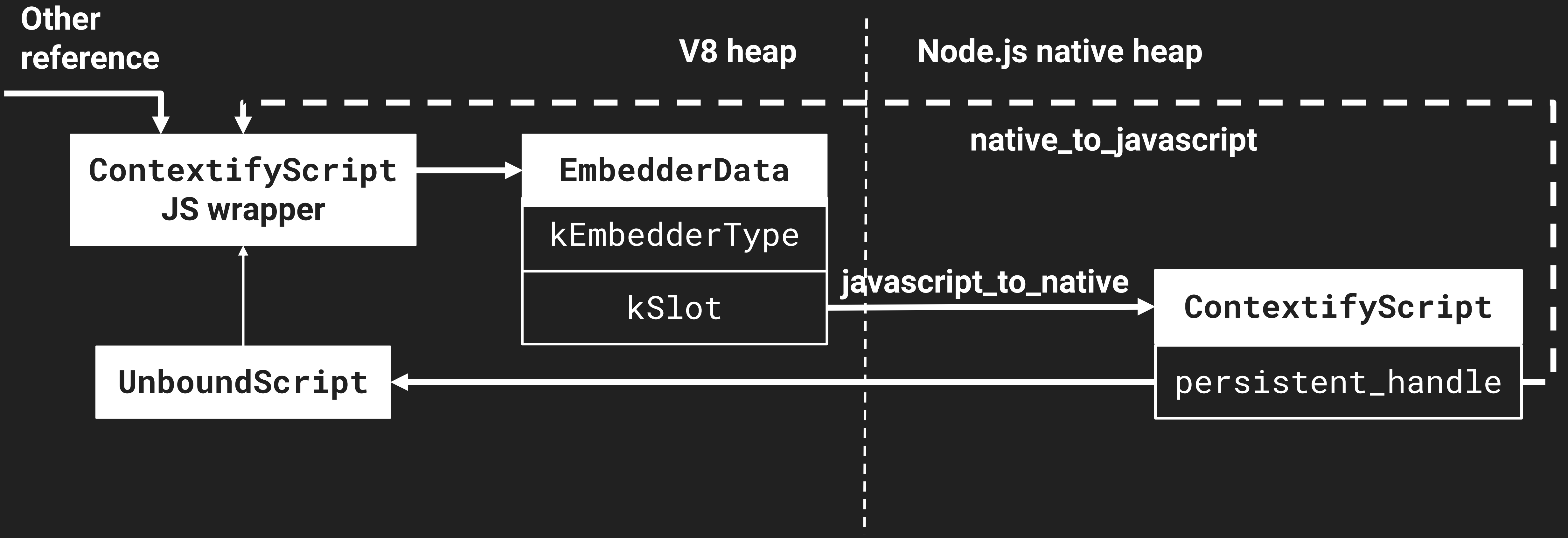
Case study of dynamic import() memory issues

```
class ContextifyScript : public BaseObject {
 public:
 ContextifyScript(Environment* env, Local<Object> object)
   : BaseObject(env, object) {
   script_.Reset(isolate, v8_script);
   MakeWeak();
 }
 private:
 v8::Global<v8::UnboundScript> script_;
}
```

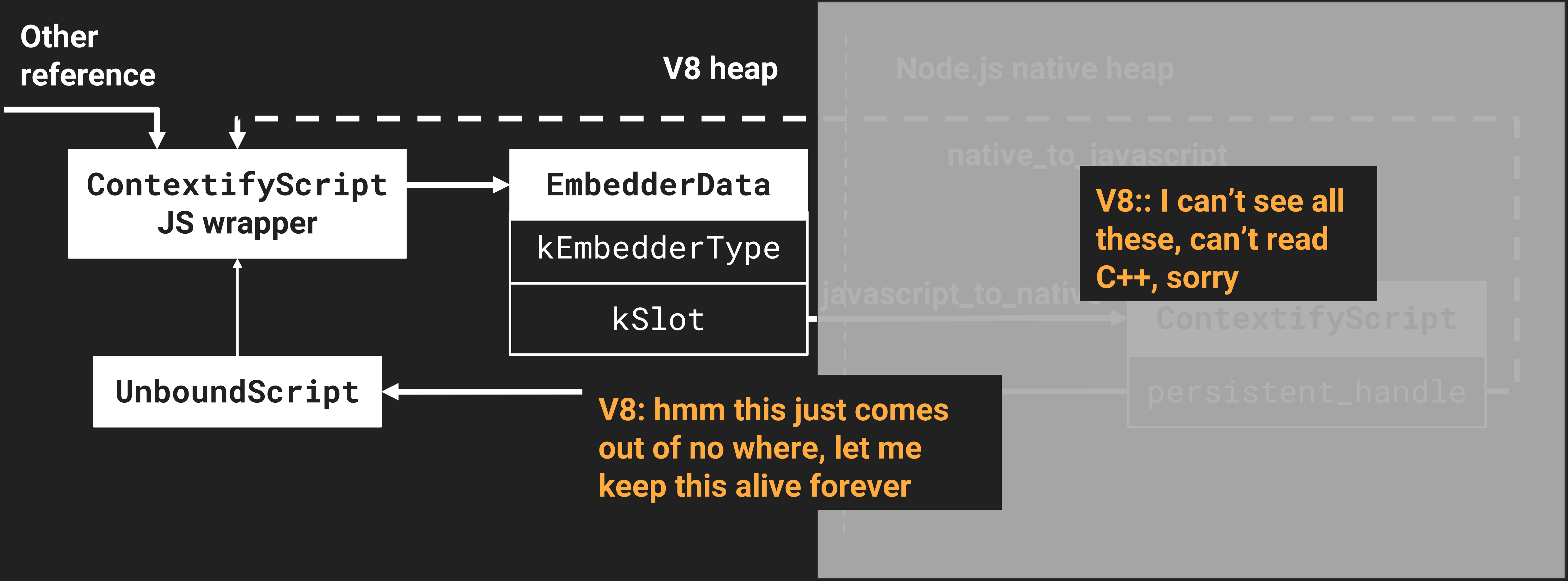
After fixing JS land reference to ensure ContextifyScript is kept alive while import() can still be called from the Script…

😅 This started to leak because the UnboundScript kept ContextifyScript alive in a cycle (similar to #42080)

Case study of dynamic import() memory issues



Case study of dynamic import() memory issues



V8:: I can't see all these, can't read C++, sorry

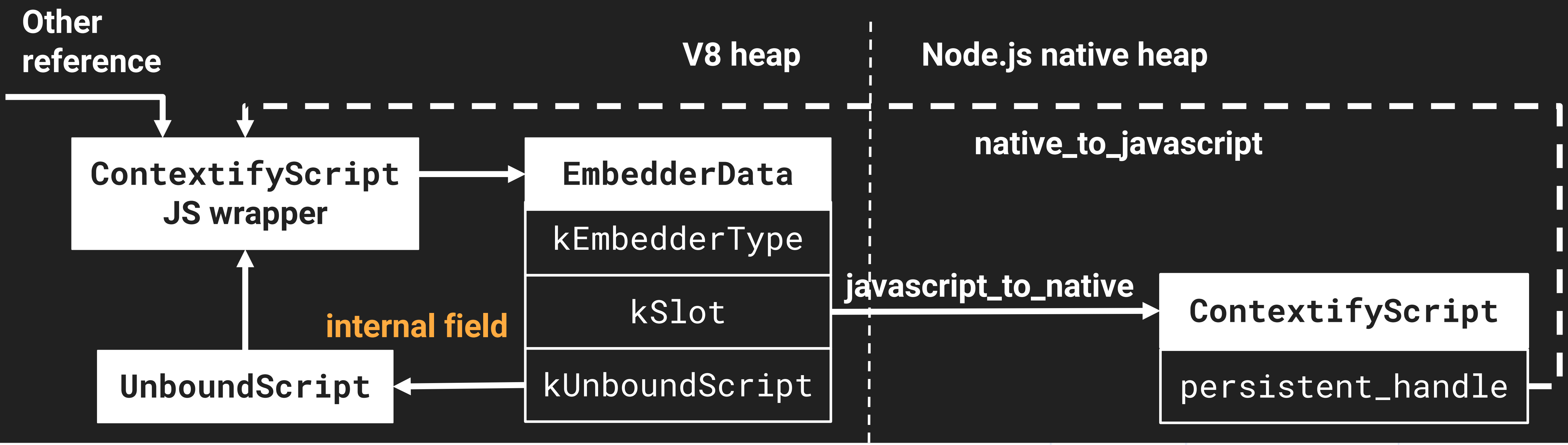
V8: hmm this just comes out of no where, let me keep this alive forever

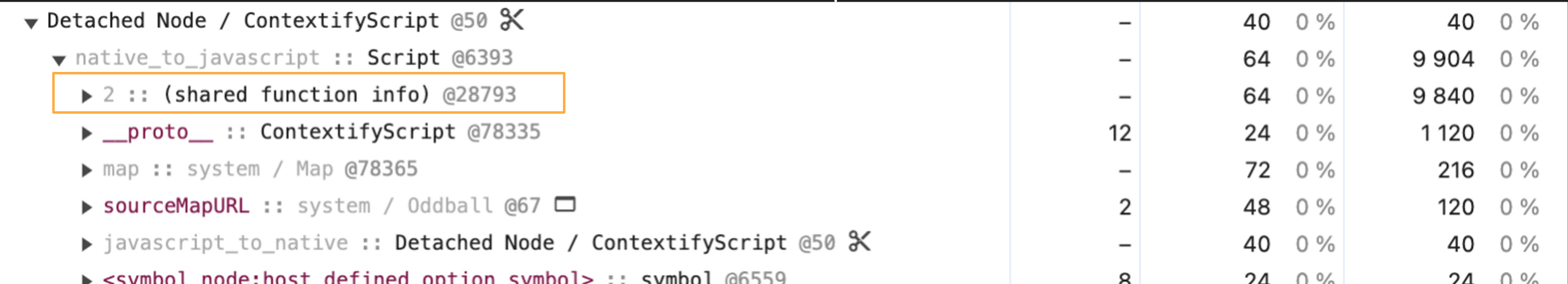
Case study of dynamic import() memory issues

```
class ContextifyScript : public BaseObject {
 public:
 ContextifyScript(Environment* env, Local<Object> object)
   : BaseObject(env, object) {
   script_.Reset(isolate, v8_script);
   MakeWeak();
   script_.SetWeak();
   object->SetInternalField(kUnboundScriptSlot, v8_script);
 }
 private:
  v8::Global<v8::UnboundScript> script_;
}
```

Patched V8 a bit to make this possible to fix the leak

Case study of dynamic import() memory issues



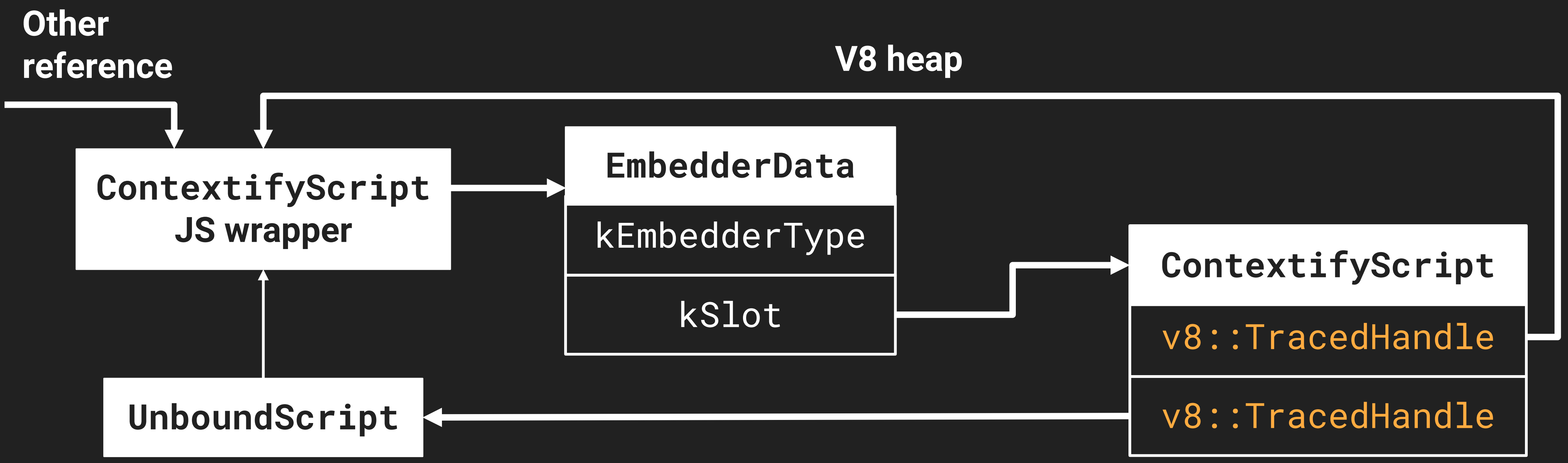


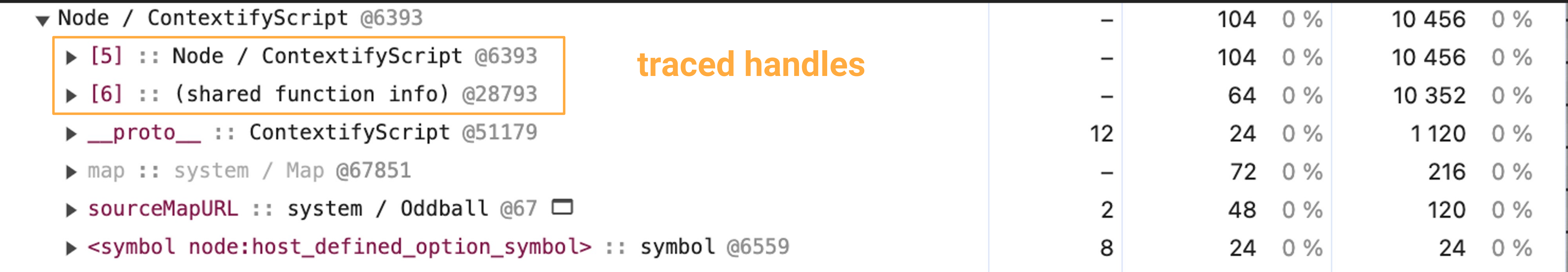
```
▼ Detached Node / ContextifyScript @50 ✂                    –    40  0 %     40  0 %
  ▼ native_to_javascript :: Script @6393                    –    64  0 %  9 904  0 %
    ▶ 2 :: (shared function info) @28793                    –    64  0 %  9 840  0 %
    ▶ __proto__ :: ContextifyScript @78335                 12    24  0 %  1 120  0 %
    ▶ map :: system / Map @78365                            –    72  0 %    216  0 %
    ▶ sourceMapURL :: system / Oddball @67 ▭                2    48  0 %    120  0 %
    ▶ javascript_to_native :: Detached Node / ContextifyScript @50 ✂   –    40  0 %     40  0 %
    ▶ <symbol node:host_defined_option_symbol> :: symbol @6559   8    24  0 %     24  0 %
```

A path to Oilpan

- V8’s C++ GC library
- Can be used to create C++ <-> JS references that V8 knows how to track to avoid issues caused by cycles
- Bootstrapped in core with some cctests
- Currently working on migration plan (design doc)

A path to Oilpan (WIP PR: #52295)





```
▼ Node / ContextifyScript @6393                                      –   104  0 %   10 456  0 %
   ▶ [5] :: Node / ContextifyScript @6393                            –   104  0 %   10 456  0 %
   ▶ [6] :: (shared function info) @28793                            –    64  0 %   10 352  0 %
   ▶ __proto__ :: ContextifyScript @51179                           12    24  0 %    1 120  0 %
   ▶ map :: system / Map @67851                                      –    72  0 %      216  0 %
   ▶ sourceMapURL :: system / Oddball @67 □                          2    48  0 %      120  0 %
   ▶ <symbol node:host_defined_option_symbol> :: symbol @6559        8    24  0 %       24  0 %
```

traced handles

A path to Oilpan (WIP PR: #52295)

- `crypto.createHash()` can be ~27% faster after migration. The whole hashing operation can be ~10% faster. #51017
- Blocker: external memory tracking in heap snapshots. Working on a new V8 API.
- Blocker: saw minor regressions in some APIs (V8 serdes), investigating impact & why.
- Blocker: Need to be able to trace v8::Data. Working on an upstream patch crrev:5403888

How to analyze memory issues

- Good old heap snapshots: `v8.writeHeapSnapshot()` or `--heapsnapshot-near-heap-limit`
- `--heap-profiler-show-hidden-objects`
- `--heap-prof`
- `vm.measureMemory()` etc. can provide hints
- Sometimes llnode helps too
- Discuss: what else can we provide? What are some common memory issues that can use better tooling support from Node.js?