Checking Node Add-ons for Memory Leaks using Valgrind

Valgrind is an open-source memory debugging and profiling tool for Linux-based operating systems. It is designed to help developers find memory-related problems in their code. Valgrind operates by running the program being tested in a virtual environment, simulating a CPU and memory system. This allows it to track all memory accesses, detect common errors such as memory leaks, uninitialized memory, buffer overflows, and other memory-related bugs that can be difficult to find and fix manually.

Using Valgrind

This guide uses a node-addon-api add-on from the node-addon-examples repository: https://github.com/nodejs/node-addon-examples/tree/5464927/6_object_wrap/node-addon-api.

Clone the repository and build the sample in the 6_object_wrap/node-addon-api folder:

```
cd 6_object_wrap/node-addon-api && npm i
```

Run the example via `valgrind`:

```
valgrind --leak-check=full node addon.js
```

Valgrind produces a report when the process exits:

```
==12584==
HEAP SUMMARY:
   in use at exit: 11,491 bytes in 32 blocks
 total heap usage: 21,739 allocs, 21,707 frees, 19,012,068 bytes allocated
==12584==
64 bytes in 1 blocks are definitely lost in loss record 13 of 32
  at 0x5074E63: operator new(unsigned long) (in
/usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
  by 0xB3E975: napi_module_register (in /home/kevin/.nvm/versions/node/v18.14.2/bin/node)
  by 0x504AB99: call_init.part.0 (dl-init.c:72)
  by 0x504ACA0: call_init (dl-init.c:30)
  by 0x504ACA0: _dl_init (dl-init.c:119)
  by 0x5DB6984: _dl_catch_exception (dl-error-skeleton.c:182)
  by 0x504F0CE: dl_open_worker (dl-open.c:758)
  by 0x5DB6927: _dl_catch_exception (dl-error-skeleton.c:208)
  by 0x504E609: _dl_open (dl-open.c:837)
  by 0x509134B: dlopen_doit (dlopen.c:66)
  by 0x5DB6927: _dl_catch_exception (dl-error-skeleton.c:208)
  by 0x5DB69F2: _dl_catch_error (dl-error-skeleton.c:227)
  by 0x5091B58: _dlerror_run (dlerror.c:170)
==12584==
304 bytes in 1 blocks are possibly lost in loss record 26 of 32
  at 0x5076D99: calloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
  by 0x504D9DA: allocate_dtv (dl-tls.c:286)
  by 0x504D9DA: _dl_allocate_tls (dl-tls.c:532)
  by 0x5209322: allocate_stack (allocatestack.c:622)
  by 0x5209322: pthread_create@@GLIBC_2.2.5 (pthread_create.c:660)
  by 0xC8C57D: node::inspector::Agent::Start(std::__cxx11::basic_string<char, std::char_traits<char>,
std::allocator<char> > const&, node::DebugOptions const&,
std::shared_ptr<node::ExclusiveAccess<node::HostPort, node::MutexBase<node::LibuvMutexTraits> > >, bool)
(in /home/kevin/.nvm/versions/node/v18.14.2/bin/node)
  by 0xB37554:
node::Environment::InitializeInspector(std::unique_ptr<node::inspector::ParentInspectorHandle,
std::default_delete<node::inspector::ParentInspectorHandle> >) (in
/home/kevin/.nvm/versions/node/v18.14.2/bin/node)
  by 0xBC0FDF: node::NodeMainInstance::CreateMainEnvironment(int*) (in
/home/kevin/.nvm/versions/node/v18.14.2/bin/node)
  by 0xBC122C: node::NodeMainInstance::Run() (in /home/kevin/.nvm/versions/node/v18.14.2/bin/node)
  by 0xB364F3: node::LoadSnapshotDataAndRun(node::SnapshotData const**, node::InitializationResult const*)
(in /home/kevin/.nvm/versions/node/v18.14.2/bin/node)
  by 0xB3A0EE: node::Start(int, char**) (in /home/kevin/.nvm/versions/node/v18.14.2/bin/node)
  by 0x5C7A082: (below main) (libc-start.c:308)
==12584==
LEAK SUMMARY:
  definitely lost: 64 bytes in 1 blocks
  indirectly lost: 0 bytes in 0 blocks
    possibly lost: 304 bytes in 1 blocks
  still reachable: 11,123 bytes in 30 blocks
       suppressed: 0 bytes in 0 blocks
Reachable blocks (those to which a pointer was found) are not shown.
To see them, rerun with: --leak-check=full --show-leak-kinds=all
==12584==
For lists of detected and suppressed errors, rerun with: -s
ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0
```

The leak summary shows some bytes definitely lost and some bytes possibly lost:

- 64 definitely lost bytes: allocated by stack that includes `napi_module_register`. Node calls this method for each add-on loaded. Add-ons are dynamically loaded using `dlopen`, which requires some memory to store the handle to the native library. Node does not `dlclose()` successfully registered add-ons because it is impossible for Node to determine when the add-on is truly gone.
- 304 possibly lost bytes: allocated by a stack that includes `node::inspector::Agent::Start`.

It is possible to hide these acceptable leaks using the suppression feature of `valgrind`.

Generating a Valgrind Suppression File

Valgrind uses suppression files to hide issues found from the summary. Generate a log file with embedded suppressions using the `--gen-suppressions` flag:

```
valgrind --leak-check=full \
   --gen-suppressions=all \
   --log-file=./valgrind-out.txt \
   node addon.js
```

Valgrind will save the output to the log file specified. After each heap in the summary, Valgrind will include a suppression record: a structure that Valgrind can use to not report specific memory issues. Suppression records can be saved to the suppression file which Valgrind can use in subsequent executions to hide various memory errors. This is an example of the suppression records for the two heaps previously mentioned:

```
{
   <insert_a_suppression_name_here>
  Memcheck:Leak
  match-leak-kinds: definite
  fun:_Znwm
  fun:napi_module_register
  fun:call_init.part.0
  fun:call_init
  fun:_dl_init
  fun:_dl_catch_exception
  fun:dl_open_worker
  fun:_dl_catch_exception
  fun:_dl_open
  fun:dlopen_doit
  fun:_dl_catch_exception
  fun:_dl_catch_error
  fun:_dlerror_run
}
{
   <insert_a_suppression_name_here>
  Memcheck:Leak
  match-leak-kinds: possible
  fun:calloc
  fun:allocate_dtv
  fun:_dl_allocate_tls
  fun:allocate_stack
  fun:pthread_create@@GLIBC_2.2.5

fun:_ZN4node9inspector5Agent5StartERKNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEERKNS_12DebugOptionsESt10shared_ptrINS_15ExclusiveAccessINS_8HostPortENS_9MutexBaseINS_16LibuvMutexTraitsEEEEEEb

fun:_ZN4node11Environment19InitializeInspectorESt10unique_ptrINS_9inspector21ParentInspectorHandleESt14default_deleteIS3_EE
  fun:_ZN4node16NodeMainInstance21CreateMainEnvironmentEPi
  fun:_ZN4node16NodeMainInstance3RunEv
  fun:_ZN4node22LoadSnapshotDataAndRunEPPKNS_12SnapshotDataEPKNS_20InitializationResultE
  fun:_ZN4node5StartEiPPc
  fun:(below main)
}
```

Create a file (eg. `node-18.14.2.supp` with the contents of the two suppression records.

Running Valgrind with Suppressions

Run `valgrind` with the suppression file previously created:

```
valgrind --leak-check=full \
   --suppressions=./node-18.14.2.supp \
   node addon.js
```

Now, the Valgrind report will not contain the two acceptable leaks, and are only mentioned as `suppressed` in the leak summary:

```
HEAP SUMMARY:
   in use at exit: 11,491 bytes in 32 blocks
 total heap usage: 21,741 allocs, 21,709 frees, 19,015,286 bytes allocated

LEAK SUMMARY:
  definitely lost: 0 bytes in 0 blocks
  indirectly lost: 0 bytes in 0 blocks
    possibly lost: 0 bytes in 0 blocks
  still reachable: 11,123 bytes in 30 blocks
       suppressed: 368 bytes in 2 blocks
```

Example: Introducing and Fixing a Memory Leak

Let's introduce an actual memory leak in `MyObject` by changing the concrete `double value_` to a pointer:

```
myobject.h:
-  double value_;
+  double* value_;

myobject.cc:
@@ -28,17 +28,17 @@ MyObject::MyObject(const Napi::CallbackInfo& info)
  }

  Napi::Number value = info[0].As<Napi::Number>();
-  this->value_ = value.DoubleValue();
+  value_ = new double { value.DoubleValue() };
```

```
}

Napi::Value MyObject::GetValue(const Napi::CallbackInfo& info) {
-  double num = this->value_;
+  double num = *value_;

  return Napi::Number::New(info.Env(), num);
}

Napi::Value MyObject::PlusOne(const Napi::CallbackInfo& info) {
-  this->value_ = this->value_ + 1;
+  *value_ = *value_ + 1;

  return MyObject::GetValue(info);
}
@@ -52,7 +52,11 @@ Napi::Value MyObject::Multiply(const Napi::CallbackInfo& info) {
  }

  Napi::Object obj = info.Env().GetInstanceData<Napi::FunctionReference>()->New(
-      {Napi::Number::New(info.Env(), this->value_ * multiple.DoubleValue())});
+      {Napi::Number::New(info.Env(), *value_ * multiple.DoubleValue())});

  return obj;
}
```

Valgrind will report bytes definitely lost at a stack trace that includes the `MyObject` constructor, where creating a `new double` takes place:

```
8 bytes in 1 blocks are definitely lost in loss record 2 of 34
  at 0x5074E63: operator new(unsigned long) (in
/usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
  by 0x11A59246: MyObject::MyObject(Napi::CallbackInfo const&) (in
/home/kevin/Projects/node-addon-examples/6_object_wrap/node-addon-api/build/Release/addon.node)
  by 0x11A60871: Napi::ObjectWrap<MyObject>::ConstructorCallbackWrapper(napi_env__*,
napi_callback_info__*) (in
/home/kevin/Projects/node-addon-examples/6_object_wrap/node-addon-api/build/Release/addon.node)
  by 0xB1F668: v8impl::(anonymous
namespace)::FunctionCallbackWrapper::Invoke(v8::FunctionCallbackInfo<v8::Value> const&) (in
/home/kevin/.nvm/versions/node/v18.14.2/bin/node)

LEAK SUMMARY:
  definitely lost: 32 bytes in 4 blocks
  indirectly lost: 0 bytes in 0 blocks
    possibly lost: 0 bytes in 0 blocks
  still reachable: 11,123 bytes in 30 blocks
       suppressed: 368 bytes in 2 blocks
```

We can solve this leak by creating a destructor for `MyObject`:

```
# myobject.h
@@ -7,13 +7,14 @@ class MyObject : public Napi::ObjectWrap<MyObject> {
 public:
  static Napi::Object Init(Napi::Env env, Napi::Object exports);
  MyObject(const Napi::CallbackInfo& info);
+  ~MyObject();

# myobject.c
+
+MyObject::~MyObject() {
+  delete value_;
+}
```

Running the same `valgrind` command now shows no unexpected memory errors:

```
LEAK SUMMARY:
  definitely lost: 0 bytes in 0 blocks
  indirectly lost: 0 bytes in 0 blocks
    possibly lost: 0 bytes in 0 blocks
  still reachable: 11,123 bytes in 30 blocks
       suppressed: 368 bytes in 2 blocks
```