Getting Started#

Valgrind is a framework that allows for dynamic instrumentation of executables. There are many useful tools built on top of it that will help with things like memory errors, threading errors, and profiling. If you are not familiar with Valgrind already, or want to know more about it, you can check out the user manual at http://valgrind.org/docs/manual/manual.html. It contains tons of useful information. This guide will cover some of the basics on how to get things up and running on the QNX Neutrino platform using the default tool, memcheck.

Supported Architectures#

We currently support Valgrind on x86 and armle-v7.

Getting Valgrind#

There is a binary release available, which would be the easiest method, but for those interested, the source code is also available for building locally.
To build Valgrind locally you will need a Software Development Platform (SDP) installed, and a non-Windows host. Verify your environment is configured to use the SDP, go to the build for your target (ie. valgrind/nto-x86-o) and type 'make install'. If all goes well, valgrind is now in your install root directory.
Once you have a copy of Valgrind you need to get it onto your target hardware. This can be done in the same way that you are getting other files there, and will depend on the your preferences as well as the limitations of your setup. Some possibilities would be to add them when generating your persistent storage (embedded flash, eide, etc), copy the files onto an existing image (QDE, ftp, scp, etc), or access the files remotely (nfs, smb, etc). The location doesn't matter, so long as the permissions allow for the binaries to be execuatable.

Running Valgrind#

Running is very striaght forward. Whichever way you are launching your program, just prepend valgrind and its options. Let's take running ls on the command line as an example:

```
# ls
# valgrind --tool=memcheck ls
==2187281== Memcheck, a memory error detector
==2187281== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==2187281== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==2187281== Command: ls
==2187281==
==2187281==
==2187281==
==2187281== Valgrind is exiting:
==2187281==   Symbols for /proc/boot/libc.so.3 are required but not found.
==2187281==   (Suggestion: compile that binary with debug-information, or provide a separate symbol-file.)
==2187281==
==2187281==
```

If your copy of Valgrind is not in /usr, then you will also need to tell it where to find its libraries. And if you do not specify a tool, memcheck will be used by default.

```
# VALGRIND_LIB=/sdcard/lib/valgrind /sdcard/bin/valgrind ls
==2551825== Memcheck, a memory error detector
==2551825== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==2551825== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==2551825== Command: ls
==2551825==
```

```
==2551825==
==2551825==
==2551825== Valgrind is exiting:
==2551825==   Symbols for /proc/boot/libc.so.3 are required but not found.
==2551825==   (Suggestion: compile that binary with debug-information, or provide a separate symbol-file.)
==2551825==
==2551825==
#
```

This runs Valgrind with the default tool, memcheck. The first time you try to run Valgrind, you will more then likely see the same error as above, which will be explained in the next section. More information on running can be found in the example section.

Getting Symbols#

Valgrind will make use of various ELF sections containing symbol and debug information. If your program has had its symbols stripped, it can be useful to tell Valgrind where to find them. There are currently a few libraries that if loaded need to either be unstripped, or have symbol files available if they are used. If symbols are not found, you should see the error message "Symbols for <file> are required but not found". The current list of required files is:

libc.so.*
libcpp.so.*
libcpp-ne.so.*
libstdc++.so.*

Besides the required symbol files, you may find some cases where you want to load some additional ones. Having them available will allow Valgrind to give you more information regarding any errors it finds, like complete stack traces, line numbers, variables names, etc. For the same reason, it may be preferred to use a debug build of your program. Loading symbol files or debug information requires some memory overhead and in low memory embedded systems it may be worthwhile to trade off verbosity for more free memory.
The binaries in the SDP are stripped, and while it's possible to get by using just those files, it is much easier/ recommended that you grab the debug info package for your corresponding SDP These can be found on the downloads page at http://qnx.com. Here are direct links to 650sp1 and 660 debug info packages.
Our port of Valgrind will go searching for symbol files when run. This will cause some extra delays during startup, but is required since there are frequently cases where file system layouts don't match the staging / symbol directory. Take libc.so.3 for exmaple, it is normally in the IFS and its path will be /proc/boot/libc.so.3, but will be in the /lib subdirectory in the staging directory. The searching overhead can be limited by using a single method for searching. Below we will detail the different methods, in the order that they will be used, using the most commonly stripped file /proc/boot/libc.so.3. When a file is found that looks like it could be a symbol file, Valgrind will attempt to validate it by using either the debuglink or buildid. In the examples below /proc/boot/libc.so.3 does not have a debuglink, so the search will use the base name, but if it did have a debuglink then the search would use that filename instead.

Current Working Directoy#

Valgrind will always look in the current working directory for symbol files. It's always based on the basename only.

./libc.so.3

Extra DebugInfo Path#

You can use the --extra-debuginfo-path=<path> to specify a single path to look in for symbol files. This can only be specified once.

<extrapath>/libc.so.3
<extrapath>/proc/boot/libc.so.3
<extrapath>/$LD_LIBRARY_PATH/libc.so.3
<extrapath>/<_CS_LIBPATH>/libc.so.3
<extrapath>/$VALGRIND_LIB/libc.so.3
<extrapath>/$PATH/libc.so.3

QNX Target#

Like other tools, Valgrind will look for the environment variable QNX_TARGET to find the staging directory. These files are used for linking and should be unstripped, perfect for our needs.

$QNX_TARGET/libc.so.3
$QNX_TARGET/proc/boot/libc.so.3
$QNX_TARGET/$LD_LIBRARY_PATH/libc.so.3
$QNX_TARGET/<_CS_LIBPATH>/libc.so.3
$QNX_TARGET/$VALGRIND_LIB/libc.so.3
$QNX_TARGET/$PATH/libc.so.3

DebugInfo Server#

Valgrind comes with a server that allows remote access to symbol files, and should be available for your host through your operating system's packaging system of choice or by building from the official source releases. The server should be started on the host and it looks in the current working directory when files are requested.

$ cd /opt/650sdp/target/qnx6/x86
$ valgrind-di-server <port>

Then on the target you just need to tell Valgrind the address and port to connect to the server --debuginfo-server=<ip>:<port>

<serverpath>/libc.so.3
<serverpath>/proc/boot/libc.so.3
<serverpath>/$LD_LIBRARY_PATH/libc.so.3
<serverpath>/<_CS_LIBPATH>/libc.so.3
<serverpath>/$VALGRIND_LIB/libc.so.3
<serverpath>/$PATH/libc.so.3

No debuglink or buildid#

Since validation is done with debuglink or buildid, if neither of these are present then validation is not possible and Valgrind will not load your symbol files. This is the case for the 6.5.0 SDP. So long as you are absolutely sure that the correct symbol files are being found, you can use --allow-mismatched-debuginfo=yes to skip validation. When using this option it is very important to make sure that the files match, if they don't you may see assertion failures, incorrect stack traces, and all sorts of other confusing behaviour. To make sure the files are correct, the --allow-mismatched-debuginfo will not apply to the search paths listed previously, instead there are only a couple explicitly specified locations where it will look. To specify the locations you can use either --extra-debuginfo-path=<path> or --debuginfo-server=<ip>:<port> and will only consider the following paths for mismatched debuginfo:

<extrapath>/proc/boot/libc.so.3
<extrapath>/libc.so.3
<serverpath>/libc.so.3

If you have problems loading symbols or just want more information on what exactly is being loaded, you can add the -v option to increase the verbosity. If you want a very detailed list of every single path that was checked then you can use -v -v -v. As an example, let's look in an NFS mounted stage directory:

```
# QNX_TARGET=/nfs/target/qnx6 valgrind -v -v -v ls
...
--1728529-- Reading syms from /proc/boot/libc.so.3
--1728529--    svma 0x0000000000, avma 0x0000001000
--1728529--   Considering for search suffix:
--1728529--        ""
--1728529--        "/proc/boot"
--1728529--        LD_LIBRARY_PATH="(null)"
--1728529--        _CS_LIBPATH="/proc/boot:/lib:/usr/lib:/lib/dll:/opt/lib"
--1728529--        PATH="/proc/boot:/bin:/usr/bin:/opt/bin/sbin:/usr/sbin"
--1728529--   Considering for search prefix:
--1728529--        extra-debuginfo-path="(null)"
--1728529--        QNX_TARGET="/nfs/target/qnx6"
--1728529--        debuginfo-server="(null)"
--1728529--   Looking for libc.so.3 ..
--1728529--   Looking for /nfs/target/qnx6/x86//libc.so.3 ..
--1728529--   Looking for /nfs/target/qnx6/x86//proc/boot/libc.so.3 ..
--1728529--   Looking for /nfs/target/qnx6/x86//proc/boot/libc.so.3 ..
--1728529--   Looking for /nfs/target/qnx6/x86//lib/libc.so.3 ..
--1728529--   Considering /nfs/target/qnx6/x86//lib/libc.so.3 ..
--1728529--   .. build-id is valid
...
```

False Positives#

From time to time you may run across some false positives, or problems in libraries you do not have access too. Given our OS's heavy use of message passing using structs, the uninitialized members tend to be the largest source of false positives. In these cases the output may just be noise, and can be suppressed. You can specify suppression files with --suppressions=<supp-file>. The suppression entries can be hand crafted, or Valgrind can do the work with --gen-suppressions=yes. You can find more information here http://valgrind.org/docs/manual/manual-core.html#manual-core.suppress.

General Options#

Each tool has its own set of options, but there is also a set of general options that apply to all the tools. There are many of them, but there are only a couple that are commonly used.

You can redirect Valgrind's output to a file with --log-file=<file>. The filename can also contain format specifiers, which allows for things like pid based log files, --log-file=/tmp/vg.%p.log.

If your program runs self-modifying code anywhere other than the stack you may need to tell Valgrind about it with --smc-check=all-non-file.

Child processes can be traced as well with --trace-children=yes. A pid based log filename is useful with this option.

File descriptors can be tracked with --track-fds=yes.

Timestamps can be added to the output with --time-stamp=yes.

Example#

Hopefully that covered all the basics, so now we can jump into the deep end. In order to give us something interesting to look at, let's use a purposefully bad program.

```
#include <stdlib.h>
```

```
#include <stdio.h>

void* func1(void)
{
   char *ptr1 = malloc(10);
   /* do some work */
   return malloc(20);
}

void func2(void)
{
   char *ptr2 = malloc(10);
   /* do some work */
   if (ptr2[1]) {
      printf("ptr2[1] != 0\n");
   }
   /* do some more work */
   return;
}

void func3(void *ptr3)
{
   /* do some work */
   free(ptr3);
   /* do some more work */
   free(ptr3);
   return;
}

int main(void)
{
   void *ptr = func1();
   func2();
   func3(ptr);
   return 0;
}
```

Here is a basic run:

```
# valgrind ./bug
==2314257== Memcheck, a memory error detector
==2314257== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==2314257== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==2314257== Command: ./bug
==2314257==
==2314257== Conditional jump or move depends on uninitialised value(s)
==2314257==    at 0x8000738: func2 (in /valgrind/tmp/bug)
==2314257==    by 0x8000793: main (in /valgrind/tmp/bug)
==2314257==
==2314257== Invalid free() / delete / delete[] / realloc()
==2314257==    at 0xBFCD6: free (vg_replace_malloc.c:527 in /valgrind/memcheck/vgpreload_memcheck-x86-nto.so)
==2314257==    by 0x8000776: func3 (in /valgrind/tmp/bug)
==2314257==    by 0x800079F: main (in /valgrind/tmp/bug)
==2314257==  Address 0xc50b0 is 0 bytes inside a block of size 20 free'd
==2314257==    at 0xBFCD6: free (vg_replace_malloc.c:527 in /valgrind/memcheck/vgpreload_memcheck-x86-nto.so)
==2314257==    by 0x800076B: func3 (in /valgrind/tmp/bug)
==2314257==    by 0x800079F: main (in /valgrind/tmp/bug)
==2314257==
==2314257==
==2314257== HEAP SUMMARY:
```

```
==2314257==    in use at exit: 20 bytes in 2 blocks
==2314257==  total heap usage: 4 allocs, 3 frees, 64 bytes allocated
==2314257==
==2314257== LEAK SUMMARY:
==2314257==    definitely lost: 20 bytes in 2 blocks
==2314257==    indirectly lost: 0 bytes in 0 blocks
==2314257==      possibly lost: 0 bytes in 0 blocks
==2314257==    still reachable: 0 bytes in 0 blocks
==2314257==         suppressed: 0 bytes in 0 blocks
==2314257== Rerun with --leak-check=full to see details of leaked memory
==2314257==
==2314257== For counts of detected and suppressed errors, rerun with: -v
==2314257== Use --track-origins=yes to see where uninitialised values come from
==2314257== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
#
```

The first error reported is a conditional jump or move that depends on uninitialized value. We know it's in func2(), but if that function is very large it may still be very difficult to find. If we wanted more information, the only option we have is to compile with debug information. That should give us extra info like line numbers, so let's see what the output looks like.

```
# valgrind ./bug_g
==2355217== Memcheck, a memory error detector
==2355217== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==2355217== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==2355217== Command: ./bug_g
==2355217==
==2355217== Conditional jump or move depends on uninitialised value(s)
==2355217==    at 0x8000738: func2 (bug.c:15 in /valgrind/tmp/bug)
==2355217==    by 0x8000793: main (bug.c:34 in /valgrind/tmp/bug)
==2355217==
==2355217== Invalid free() / delete / delete[] / realloc()
==2355217==    at 0xBFCD6: free (vg_replace_malloc.c:527 in /valgrind/memcheck/vgpreload_memcheck-x86-nto.so)
==2355217==    by 0x8000776: func3 (bug.c:27 in /valgrind/tmp/bug)
==2355217==    by 0x800079F: main (bug.c:35 in /valgrind/tmp/bug)
==2355217==  Address 0xc50b0 is 0 bytes inside a block of size 20 free'd
==2355217==    at 0xBFCD6: free (vg_replace_malloc.c:527 in /valgrind/memcheck/vgpreload_memcheck-x86-nto.so)
==2355217==    by 0x800076B: func3 (bug.c:25 in /valgrind/tmp/bug)
==2355217==    by 0x800079F: main (bug.c:35 in /valgrind/tmp/bug)
==2355217==
==2355217==
==2355217== HEAP SUMMARY:
==2355217==    in use at exit: 20 bytes in 2 blocks
==2355217==  total heap usage: 4 allocs, 3 frees, 64 bytes allocated
==2355217==
==2355217== LEAK SUMMARY:
==2355217==    definitely lost: 20 bytes in 2 blocks
==2355217==    indirectly lost: 0 bytes in 0 blocks
==2355217==      possibly lost: 0 bytes in 0 blocks
==2355217==    still reachable: 0 bytes in 0 blocks
==2355217==         suppressed: 0 bytes in 0 blocks
==2355217== Rerun with --leak-check=full to see details of leaked memory
==2355217==
==2355217== For counts of detected and suppressed errors, rerun with: -v
==2355217== Use --track-origins=yes to see where uninitialised values come from
==2355217== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
#
```

Now we can see that the first error takes place on line 15. It looks like after allocating and doing some work, when we hit line 15 ptr2[1] has not yet been initialized and is used by the if statement. If you wanted to know where the uninitialized data originated, then you can use the --track-origins=yes option.

```
# valgrind --track-origins=yes ./bug_g
==2457617== Memcheck, a memory error detector
==2457617== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==2457617== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==2457617== Command: ./bug_g
==2457617==
==2457617== Conditional jump or move depends on uninitialised value(s)
==2457617==    at 0x8000738: func2 (bug.c:15 in /valgrind/tmp/bug_g)
==2457617==    by 0x8000793: main (bug.c:33 in /valgrind/tmp/bug_g)
==2457617==  Uninitialised value was created by a heap allocation
==2457617==    at 0xC0CD9: malloc (vg_replace_malloc.c:306 in /valgrind/memcheck/vgpreload_memcheck-x86-nto.so)
==2457617==    by 0x800072C: func2 (bug.c:13 in /valgrind/tmp/bug_g)
==2457617==    by 0x8000793: main (bug.c:33 in /valgrind/tmp/bug_g)
==2457617==
==2457617== Invalid free() / delete / delete[] / realloc()
==2457617==    at 0xBFCD6: free (vg_replace_malloc.c:527 in /valgrind/memcheck/vgpreload_memcheck-x86-nto.so)
==2457617==    by 0x8000776: func3 (bug.c:25 in /valgrind/tmp/bug_g)
==2457617==    by 0x800079F: main (bug.c:34 in /valgrind/tmp/bug_g)
==2457617==  Address 0xc50b0 is 0 bytes inside a block of size 20 free'd
==2457617==    at 0xBFCD6: free (vg_replace_malloc.c:527 in /valgrind/memcheck/vgpreload_memcheck-x86-nto.so)
==2457617==    by 0x800076B: func3 (bug.c:25 in /valgrind/tmp/bug_g)
==2457617==    by 0x800079F: main (bug.c:34 in /valgrind/tmp/bug_g)
==2457617==
==2457617==
==2457617== HEAP SUMMARY:
==2457617==     in use at exit: 20 bytes in 2 blocks
==2457617==   total heap usage: 4 allocs, 3 frees, 64 bytes allocated
==2457617==
==2457617== LEAK SUMMARY:
==2457617==    definitely lost: 20 bytes in 2 blocks
==2457617==    indirectly lost: 0 bytes in 0 blocks
==2457617==      possibly lost: 0 bytes in 0 blocks
==2457617==    still reachable: 0 bytes in 0 blocks
==2457617==         suppressed: 0 bytes in 0 blocks
==2457617== Rerun with --leak-check=full to see details of leaked memory
==2457617==
==2457617== For counts of detected and suppressed errors, rerun with: -v
==2457617== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
#
```

The next error in the list is an attempt to free up a bad pointer. We already have the line number, and all the information needed to track this one down. After that is the memory leak. It says we leaked 20 bytes in 2 blocks, but there is no other useful information. As the output mentions, if we run with --leak-check=full then we can get much more detailed information.

```
# valgrind --track-origins=yes --leak-check=full ./bug_g
==2531345== Memcheck, a memory error detector
==2531345== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==2531345== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==2531345== Command: ./bug_g
==2531345==
==2531345== Conditional jump or move depends on uninitialised value(s)
==2531345==    at 0x8000738: func2 (bug.c:15 in /valgrind/tmp/bug_g)
==2531345==    by 0x8000793: main (bug.c:34 in /valgrind/tmp/bug_g)
==2531345==  Uninitialised value was created by a heap allocation
```

```
==2531345==    at 0xC0CD9: malloc (vg_replace_malloc.c:306 in /valgrind/memcheck/vgpreload_memcheck-x86-nto.so)
==2531345==    by 0x800072C: func2 (bug.c:13 in /valgrind/tmp/bug_g)
==2531345==    by 0x8000793: main (bug.c:34 in /valgrind/tmp/bug_g)
==2531345==
==2531345== Invalid free() / delete / delete[] / realloc()
==2531345==    at 0xBFCD6: free (vg_replace_malloc.c:527 in /valgrind/memcheck/vgpreload_memcheck-x86-nto.so)
==2531345==    by 0x8000776: func3 (bug.c:27 in /valgrind/tmp/bug_g)
==2531345==    by 0x800079F: main (bug.c:35 in /valgrind/tmp/bug_g)
==2531345==  Address 0xc50b0 is 0 bytes inside a block of size 20 free'd
==2531345==    at 0xBFCD6: free (vg_replace_malloc.c:527 in /valgrind/memcheck/vgpreload_memcheck-x86-nto.so)
==2531345==    by 0x800076B: func3 (bug.c:25 in /valgrind/tmp/bug_g)
==2531345==    by 0x800079F: main (bug.c:35 in /valgrind/tmp/bug_g)
==2531345==
==2531345==
==2531345== HEAP SUMMARY:
==2531345==     in use at exit: 20 bytes in 2 blocks
==2531345==   total heap usage: 4 allocs, 3 frees, 64 bytes allocated
==2531345==
==2531345== 10 bytes in 1 blocks are definitely lost in loss record 1 of 2
==2531345==    at 0xC0CD9: malloc (vg_replace_malloc.c:306 in /valgrind/memcheck/vgpreload_memcheck-x86-nto.so)
==2531345==    by 0x80006F8: func1 (bug.c:6 in /valgrind/tmp/bug_g)
==2531345==    by 0x800078A: main (bug.c:33 in /valgrind/tmp/bug_g)
==2531345==
==2531345== 10 bytes in 1 blocks are definitely lost in loss record 2 of 2
==2531345==    at 0xC0CD9: malloc (vg_replace_malloc.c:306 in /valgrind/memcheck/vgpreload_memcheck-x86-nto.so)
==2531345==    by 0x800072C: func2 (bug.c:13 in /valgrind/tmp/bug_g)
==2531345==    by 0x8000793: main (bug.c:34 in /valgrind/tmp/bug_g)
==2531345==
==2531345== LEAK SUMMARY:
==2531345==    definitely lost: 20 bytes in 2 blocks
==2531345==    indirectly lost: 0 bytes in 0 blocks
==2531345==      possibly lost: 0 bytes in 0 blocks
==2531345==    still reachable: 0 bytes in 0 blocks
==2531345==         suppressed: 0 bytes in 0 blocks
==2531345==
==2531345== For counts of detected and suppressed errors, rerun with: -v
==2531345== ERROR SUMMARY: 4 errors from 4 contexts (suppressed: 0 from 0)
#
```

And now we can see where the leaked blocks were allocated. Leaks are put into 5 different categories.

Category	Description
definitely lost	Nothing is referencing your memory. It is a definite leak.
indirectly lost	The head of a pointer based structure is no longer being referenced. It is a definite leak.
possibly lost	Memory looks like it's being leaked, but there is still something pointing to memory in the block. Possibly a leak.
still reachable	Something is referencing the start of the memory block. Unlikely a leak.
suppressed	Leaks that were suppressed via the suppression files.

For more information on the memcheck tool and its available options, please see the manual http://valgrind.org/docs/manual/mc-manual.html.

ARM Stack Traces#

If you find some traces go through a library that doesn't have any debug/unwind information which is causing Valgrind to give truncated stack traces, consider getting the symbol files. If that is just not possbile (ie. a 3rd party lib) you can try the --unw-stack-scan-thresh=<num> option. This will attempt to scan the stack for frames if fewer than 'num' frames were found.

Working With Hardware#

In most cases Valgrind will work just fine for hardware drivers, however it is possible that Valgrind's runtime overhead will cause it to run too slow and cause timing issues while talking to the hardware. If you hit this case, you may still be able to get things working by speeding things up. Limit the number of symbol files loaded, keep symbols locally (/dev/shmem is best for speed), and disable any unneeded default options the tool has that would cause some overhead.

Depending how the driver / service is designed, you may also have problems when dealing with interrupts. While we try our best to support interrupts, we can only simulate disabling and enabling hardware interrupts. They are simulated by ensuring no other threads in the process will run while interrupts are disabled. This means if you may have problems if you require proper synchronization between your normal threads and your ISR.

Debugging With Valgrind#

At some point is may be useful to be able to debug a program running under Valgrind. This can be done rather easily with Valgrind's gdb server, but it needs to create a FIFO in $TMPDIR and have a self hosted gdb. So be sure you have a self-hosted gdb available and $TMPDIR can handle create FIFOs. Most file systems can, but NFS and /dev/shmem seem to have trouble with it. A loopback block device can be used in a pinch.

On our platform the gdb server is disabled by default and can be enabled with --vgdb-full=full. You can then have it break after a number of errors with --vgdb-error=<num>, or at _start if this is set to 0. Depending where you break / are stepping you may find registers are not always up-to-date. --vex-iropt-register-updates=allregs-at-each-insn will ensure the registers are completely updated while stepping through the code, but will incur a very large performance hit.

Once Valgrind is running it will prompt you with directions on how to connect gdb in another terminal.

```
# gdb /path/to/exe
...
(gdb) target remote | /path/to/vgdb
```