

Book **Discussion**

Read

Edit


View history

Search



# Linux Applications Debugging Techniques/Leaks

[< Linux Applications Debugging Techniques](#)

 This page may need to be [reviewed](#) for quality.

## Contents

- 1 What to look for
- 2 Valgrind
- 3 mudflap
- 4 DIY:libmtrace
  - 4.1 The basics
  - 4.2 A couple of improvements
  - 4.3 The stack trace
  - 4.4 Caveats
  - 4.5 What we got
  - 4.6 File and line
  - 4.7 Resources
- 5 DIY:libmemleak
  - 5.1 Operation
  - 5.2 Sample report
- 6 mallinfo
  - 6.1 References
- 7 /proc
  - 7.1 References
- 8 Various tools

## What to look for [\[ edit \]](#)

Memory can be allocated through many API calls:

1. `malloc()`
2. `memalign()`
3. `realloc()`
4. `mmap()`
5. `brk()` / `sbrk()`

To return memory to the OS:

1. `free()`
2. `munmap()`

## Valgrind [\[ edit \]](#)

[Valgrind](#) should be the first stop for any memory related issue. However:

1. it slows down the program by at least one order of magnitude. In particular server C++ programs can be slowed down 15-20 times.
2. from experience, some versions might have difficulties tracking `mmap()` allocated memory.
3. on amd64, the vex disassembler is likely to fail (v3.7) rather sooner than later so valgrind is of no use for any medium or intensive usage.
4. you need to write suppressions to filter down the issues reported.

If these are real drawbacks, lighter solutions are available.

```
LD_LIBRARY_PATH=/path/to/valgrind/libs:$LD_LIBRARY_PATH /path/to/valgrind
-v \
--error-limit=no \
--num-callers=40 \
```

[Main Page](#)

[Help](#)

[Browse wiki](#)

[Cookbook](#)

[Wikijunior](#)

[Featured books](#)

[Recent changes](#)

[Donations](#)

[Random book](#)

[Using Wikibooks](#)

Community

[Reading room](#)

[Community portal](#)

[Bulletin Board](#)

[Help out!](#)

[Policies and guidelines](#)

[Contact us](#)

Tools

[What links here](#)

[Related changes](#)

[Upload file](#)


[Special pages](#)

[Permanent link](#)

[Page information](#)

[Cite this page](#)

In other languages

 [Add links](#)

Sister projects

[Wikipedia](#)

[Wikiversity](#)

[Wiktionary](#)

[Wikiquote](#)

[Wikisource](#)

[Wikinews](#)

[Wikivoyage](#)

[Commons](#)

[Wikidata](#)

Print/export

[Create a collection](#)

[Download as PDF](#)

[Printable version](#)

```
--fullpath-after= \
--track-origins=yes \
--log-file=/path/to/valgrind.log \
--leak-check=full \
--show-reachable=yes \
--vex-iropt-precise-memory-exns=yes \
/path/to/program program-args
```

## mudflap [\[edit\]](#)

- [http://gcc.gnu.org/wiki/Mudflap\\_Pointer\\_Debugging](http://gcc.gnu.org/wiki/Mudflap_Pointer_Debugging)

Note: Mudflap has been removed from GCC 4.9 and later <sup>[1]</sup>

## DIY:libmtrace [\[edit\]](#)

The GNU C library comes with a built-in functionality to help detecting memory issues: `mtrace()`. One of the shortcomings: it does not log the call stacks of the memory allocations it tracks. We can build an interposition library to augment `mtrace()`.

### The basics [\[edit\]](#)

The malloc implementation in the GNU C library provides a simple but powerful way to detect memory leaks and obtain some information to find the location where the leaks occurs, and this, with rather minimal speed penalties for the program.

Getting started is as simple as it can be:

- `#include mcheck.h` in your code.
- Call `mtrace()` to install hooks for `malloc()`, `realloc()`, `free()` and `memalign()`. From this point on, all memory manipulations by these functions will be tracked. Note there are other untracked ways to allocate memory.
- Call `muntrace()` to uninstall the tracking handlers.
- Recompile.

```
#include <mcheck.h>
...
21 mtrace();
...
25 std::string* pstr = new std::string("leak");
...
27 char *leak = (char*)malloc(1024);
...
32 muntrace();
...
```

Under the hood, `mtrace()` installs the four hooks mentioned above. The information collected through the hooks is written to a log file.

**Note:** there are other ways to allocate memory, notably `mmap()`. These allocations will not be reported, unfortunately.

Next:

- Set the `MALLOC_TRACE` environment variable with the memory log name.
- Run the program.
- Run the memory log through `mtrace`.

```
$ MALLOC_TRACE=logs/mtrace.plain.log ./dleaker
$ mtrace dleaker logs/mtrace.plain.log > logs/mtrace.plain.leaks.log
$ cat logs/mtrace.plain.leaks.log
```

Memory not freed:

-----

Address	Size	Caller
0x081e2390	0x4	at 0x400fa727
0x081e23a0	0x11	at 0x400fa727
0x081e23b8	0x400	at /home/amelinte/projects/articole/memtrace/memtrace.v3/main.cpp:27

One of the leaks (the `malloc()` call) was precisely traced to the exact file and line number. However, the other leaks at

line 25, while detected, we do not know where they occur. The two memory allocations for the `std::string` are buried deep inside the C++ library. We would need the stack trace for these two leaks to pinpoint the place in **our** code.

We can use GDB (or the [trace\\_call macro](#)) to get the allocations' stacks:

```
$ gdb ./dleaker
...
(gdb) set env MALLOC_TRACE=./logs/gdb.mtrace.log

(gdb) b __libc_malloc
Make breakpoint pending on future shared library load? (y or [n])
Breakpoint 1 (__libc_malloc) pending.

(gdb) run
Starting program: /home/amelinte/projects/articole/memtrace/memtrace.v3/dleaker
Breakpoint 2 at 0xb7cf28d6
Pending breakpoint "__libc_malloc" resolved

Breakpoint 2, 0xb7cf28d6 in malloc () from /lib/i686/cmov/libc.so.6
(gdb) command
Type commands for when breakpoint 2 is hit, one per line.
End with a line saying just "end".
>bt
>cont
>end
(gdb) c
Continuing.

...

Breakpoint 2, 0xb7cf28d6 in malloc () from /lib/i686/cmov/libc.so.6
#0 0xb7cf28d6 in malloc () from /lib/i686/cmov/libc.so.6
#1 0xb7ebb727 in operator new () from /usr/lib/libstdc++.so.6
#2 0x08048a14 in main () at main.cpp:25          <== new std::string("leak");
...
Breakpoint 2, 0xb7cf28d6 in malloc () from /lib/i686/cmov/libc.so.6
#0 0xb7cf28d6 in malloc () from /lib/i686/cmov/libc.so.6
#1 0xb7ebb727 in operator new () from /usr/lib/libstdc++.so.6  <== mangled: _Znwj
#2 0xb7e95c01 in std::string::_Rep::_S_create () from /usr/lib/libstdc++.so.6
#3 0xb7e96f05 in ?? () from /usr/lib/libstdc++.so.6
#4 0xb7e970b7 in std::basic_string<char, std::char_traits<char>, std::allocator<char>
>::basic_string () from /usr/lib/libstdc++.so.6
#5 0x08048a58 in main () at main.cpp:25          <== new std::string("leak");
...

Breakpoint 2, 0xb7cf28d6 in malloc () from /lib/i686/cmov/libc.so.6
#0 0xb7cf28d6 in malloc () from /lib/i686/cmov/libc.so.6
#1 0x08048a75 in main () at main.cpp:27          <== malloc(leak);
```

### A couple of improvements [\[edit\]](#)

It would be good to have `mtrace()` itself dump the allocation stack and dispense with `gdb`. The modified `mtrace()` would have to supplement the information with:

- The stack trace for each allocation.
- Demangled function names.
- File name and line number.

Additionally, we can put the code in a library, to free the program from being instrumented with `mtrace()`. In this case, all we have to do is interpose the library when we want to trace memory allocations (and pay the performance price).

Note: getting all this information at runtime, particularly in a human-readable form will have a performance impact on the program, unlike the plain vanilla `mtrace()` supplied with `glibc`.

### The stack trace [\[edit\]](#)

A good start would be to use another API function: `backtrace_symbols_fd()`. This would print the stack directly to the log file. Perfect for a C program but C++ symbols are mangled:

```
@ /usr/lib/libstdc++.so.6:(_Znwj+27) [0xb7f1f727] + 0x9d3f3b0 0x4
**[ Stack: 8
./a.out(__gxx_personality_v0+0x304) [0x80492c8]
./a.out[0x80496c1]
./a.out[0x8049a0f]
/lib/i686/cmov/libc.so.6(__libc_malloc+0x35) [0xb7d56905]
/usr/lib/lib/libstdc++.so.6(_Znwj+0x27) [0xb7f1f727] <== here
./a.out(main+0x64) [0x8049b50]
/lib/i686/cmov/libc.so.6(__libc_start_main+0xe0) [0xb7cff450]
./a.out(__gxx_personality_v0+0x6d) [0x8049031]
**] Stack
```

For C++ we would have to get the stack (`backtrace_symbols()`), resolve each address (`dladdr()`) and demangle each symbol name (`abi::__cxa_demangle()`).

#### Caveats [\[edit\]](#)

- Memory allocation is one of these basic operation everything else builds on. One needs to allocate memory to load libraries and executables; needs to allocate memory to track memory allocations; and we hook onto it very early in the life of a process: the first pre-loaded library is the memory tracking library. Thus, any API call we make inside this interposition library can reserve surprises, especially in multi-threaded environments.
- The API functions we use to trace the stack can allocate memory. These allocations are also going through the hooks we installed. As we trace the new allocation, the hooks are activated again and another allocation is made as we trace this new allocation. We will run out of stack in this infinite loop. We break out of this pitfall by using a per-thread flag.
- The API functions we use to trace the stack can deadlock. Suppose we would use a lock while in our trace. We lock the trace lock and we call `dladdr()` which in turn tries to lock a dynamic linker internal lock. If on another thread `dlopen()` is called while we trace, `dlopen()` locks the same linker lock, then allocates memory: this will trigger the memory hooks and we now have the `dlopen()` thread wait on the trace lock with the linker lock taken. Deadlock.
- On some platforms (gcc 4.7.2 amd64) TLS calls would trip the `memalign` hook. This could result in an infinite recursion if the `memalign` hook in its turn, accesses a TLS variable.

#### What we got [\[edit\]](#)

Let's try again with our new library:

```
$ MALLOC_TRACE=logs/mtrace.stack.log LD_PRELOAD=./libmtrace.so ./dleaker
$ mtrace dleaker logs/mtrace.stack.log > logs/mtrace.stack.leaks.log
$ cat logs/mtrace.stack.leaks.log

Memory not freed:
-----
      Address      Size      Caller
0x08bf89b0         0x4 at 0x400ff727 <== here
0x08bf89e8        0x11 at 0x400ff727
0x08bf8a00       0x400 at /home/amelinte/projects/articole/memtrace/memtrace.v3/main.cpp:27
```

Apparently, not much of an improvement: the summary still does not get us back to line 25 in `main.cpp`. However, if we search for address `8bf89b0` in the trace log, we find this:

```
@ /usr/lib/libstdc++.so.6:(_Znwj+27) [0x400ff727] + 0x8bf89b0 0x4 <== here
**[ Stack: 8
[0x40022251] (./libmtrace.so+40022251)
[0x40022b43] (./libmtrace.so+40022b43)
[0x400231e8] (./libmtrace.so+400231e8)
[0x401cf905] __libc_malloc (/lib/i686/cmov/libc.so.6+35)
[0x400ff727] operator new(unsigned int) (/usr/lib/libstdc++.so.6+27) <== was: _Znwj
[0x80489cf] __gxx_personality_v0 (./dleaker+27f)
[0x40178450] __libc_start_main (/lib/i686/cmov/libc.so.6+e0) <== here
[0x8048791] __gxx_personality_v0 (./dleaker+41)
**] Stack
```

This is good, but having file and line information would be better.

## File and line [\[ edit \]](#)

Here we have a few possibilities:

- Run the address (e.g. 0x40178450 above) through the `addr2line` tool. If the address is in a shared object that the program loaded, it might not resolve properly.
- If we have a core dump of the program, we can ask `gdb` to resolve the address. Or we can attach to the running program and resolve the address.
- [Use the API described here](#). The downside is that it takes a quite heavy toll on the performance of the program.

## Resources [\[ edit \]](#)

- [The GNU C library manual](#)
- [Using libbfd](#)
- [Linux Programming Toolkit](#)

## DIY:libmemleak [\[ edit \]](#)

Building on `libmtrace`, we can go one step further and have an interposition library track the memory allocations made by the program. The library generates a report on demand, much like `Valgrind` does.

`Libmemleak` is significantly faster than `valgrind` but also has limited functionality (only leak detection).

## Operation [\[ edit \]](#)

`libmemtrace` has two drawbacks:

- The log file will quickly reach gigs
- You are left grepping the log to figure out what leaks when

A better solution would be to have an interposition library to collect memory operations information and to generate a report on-demand.

For `mmap()`/`munmap()` we have no choice but hook these directly. Thus, an call from within the application would first hit the hooks in `libmemleak`, then go to `libc`. For `malloc()`/`realloc()`/`memalign()`/`free()` we have two options:

- Use `mtrace()`/`muntrace()` as before, to install hooks that will be called from within `libc`. Thus, a `malloc()` call would first go through `libc` which will then call the hooks in `libmemtrace`. This leave us at the mercy of `libc`.
- The second solution is to hook these like `m(un)map`.

The second solution also frees `mtrace()`/`muntrace()` for on-demand report generation:

- A first call to `mtrace()` will kick in data collection.
- Subsequent calls to `mtrace()` will generate reports.
- `muntrace()` will stop data collection and will generate a final report.
- `MALLOC_TRACE` is not needed.

The application can then sprinkle its code with `mtrace()` calls at strategic places to avoid reporting much noise. These calls will do nothing in a normal operation as long as `MALLOC_TRACE` is not set. Or, the application can be completely ignorant of the ongoing data collection (no `mtrace()` calls within the application code) and `libmemleak` can start collecting as early as being loaded and generate one report upon being unloaded.

To control the `libmemleak` functionality, an environment variable - `MEMLEAK_CONFIG` - has to be set before loading the library:

```
export MEMLEAK_CONFIG=mtraceinit
```

- `mtraceinit` will instruct the library to start collecting data upon being loaded. Default is off and the application has to be instrumented with `m(un)trace` calls.

Thus, all the hooks have to do is to call into the reporting:

```
extern "C" void *__libc_malloc(size_t size);
extern "C" void *malloc(size_t size)
{
    void *ptr = __libc_malloc(size);
    if ( _capture_on ) {
        libmemleak::alloc(ptr, size);
    }
}
```

```

    }

    return ptr;
}

extern "C" void __libc_free(void *ptr);
extern "C" void free(void *ptr)
{
    __libc_free(ptr);
    if (_capture_on) {
        libmemleak::free(ptr, 0); // Call to reporting
    }
    else {
        serror(ptr, "Untraced free", __FILE__, __LINE__);
    }
}

extern "C" void mtrace ()
{
    // Make sure not to track memory when globals get destructed
    static std::atomic<bool> _atexit(false);
    if (!_atexit.load(std::memory_order_acquire)) {
        int ret = atexit(muntrace);
        assert(0 == ret);
        _atexit.store(true, std::memory_order_release);
    }

    if (!_capture_on) {
        _capture_on = true;
    }
    else {
        libmemleak::report();
    }
}

```

#### Sample report [\[edit\]](#)

```

// Leaks since previous report
=====

// Ordered by Num Total Bytes
// Stack Key,  Num Total Bytes,  Num Allocs,  Num Delta Bytes
5163ae4c,    1514697,           5000,      42420
...

11539977 total bytes, since previous report: 42420 bytes
Max tracked: stacks=6, allocations=25011

// All known allocations
=====

// Key  Total Bytes  Allocations
4945512: 84983 bytes in 5000 allocations
bbc54f2: 1029798 bytes in 10000 allocations
...

bbc54f2: 1029798 bytes in 10000 allocations
[0x4005286a]
lpt::stack::detail::call_stack<lpt::stack::bfd::source_resolver>::call_stack()
(binaries/lib/libmemleak_mtrace_hooks.so+0x66) in crtstuff.c:0
[0x4005238d] _pstack::_pstack() (binaries/lib/libmemleak_mtrace_hooks.so+0x4b) in
crtstuff.c:0
[0x4004f8dd] libmemleak::alloc(void*, unsigned long long)
(binaries/lib/libmemleak_mtrace_hooks.so+0x75) in crtstuff.c:0
[0x4004ee7c] ?? (binaries/lib/libmemleak_mtrace_hooks.so+0x4004ee7c) in crtstuff.c:0
[0x402f5905] ?? (/lib/i686/cmov/libc.so.6+0x35) in ??:0
[0x401a02b7] operator new(unsigned int) (/opt/lpt/gcc-4.7.0-bin/lib/libstdc++.so.6+0x27)
in crtstuff.c:0
[0x8048e3b] ?? (binaries/bin/1001leakseach+0x323) in

```

```

/home/amelinte/projects/articole/lpt/lpt/tests/1001leakseach.cpp:68
[0x8048e48] ?? (binaries/bin/1001leakseach+0x330) in
/home/amelinte/projects/articole/lpt/lpt/tests/1001leakseach.cpp:74
[0x8048e61] ?? (binaries/bin/1001leakseach+0x349) in
/home/amelinte/projects/articole/lpt/lpt/tests/1001leakseach.cpp:82
[0x8048eab] ?? (binaries/bin/1001leakseach+0x393) in
/home/amelinte/projects/articole/lpt/lpt/tests/1001leakseach.cpp:90
[0x401404fb] ?? (/lib/i686/cmov/libpthread.so.0+0x401404fb) in ??:0
[0x4035e60e] ?? (/lib/i686/cmov/libc.so.6+0x5e) in ??:0

```

*// Crosstalk: leaked bytes per stack frame*

```

-----
1029798 bytes: [0x8048e3b] ?? (binaries/bin/1001leakseach+0x323) in
/home/amelinte/projects/articole/lpt/lpt/tests/1001leakseach.cpp:68
...

```

*// Mem Address, Stack Key, Bytes*

```

-----
0x8ce7988,    bbc54f2,    4
...

```

This report took 44 ms to generate.

## mallinfo [\[edit\]](#)

The `mallinfo()` API is rumored to be deprecated. But, if available, it is very useful:

```

#include <malloc.h>

namespace process {

class memory
{
public:

    memory() : _meminfo::_mallinfo() {}

    int total() const
    {
        return _meminfo.hblkhd + _meminfo.uordblks;
    }

    bool operator==(memory const& other) const
    {
        return total() == other.total();
    }

    bool operator!=(memory const& other) const
    {
        return total() != other.total();
    }

    bool operator<(memory const& other) const
    {
        return total() < other.total();
    }

    bool operator<=(memory const& other) const
    {
        return total() <= other.total();
    }

    bool operator>(memory const& other) const
    {
        return total() > other.total();
    }

    bool operator>=(memory const& other) const
    {

```

```

        return total() >= other.total();
    }

private:

    struct mallinfo _meminfo;
};

} //process

```

```

#include <iostream>
#include <string>
#include <cassert>

int main()
{
    process::memory first;

    {
        void* p = ::malloc(1025);
        process::memory second;
        std::cout << "Mem diff: " << second.total() - first.total() << std::endl;
        assert(second > first);

        ::free(p);
        process::memory third;
        std::cout << "Mem diff: " << third.total() - first.total() << std::endl;
        assert(third == first);
    }
    {
        std::string s("abc");
        process::memory second;
        std::cout << "Mem diff: " << second.total() - first.total() << std::endl;
        assert(second > first);
    }

    process::memory fourth;
    assert(first == fourth);

    return 0;
}

```

## References [\[edit\]](#)

- [mallinfo](#)
- [mallinfo deprecated](#)

## /proc [\[edit\]](#)

Coarse grained information can be obtained from /proc:

```

#!/bin/ksh
#
# Based on:
# http://stackoverflow.com/questions/131303/linux-how-to-measure-actual-memory-usage-of-
# an-application-or-process
#
# Returns total memory used by process $1 in kb.
#
# See /proc/PID/smmaps; This file is only present if the CONFIG_MMU
# kernel configuration option is enabled
#

IFS=$'\n'

for line in $(</proc/$1/smmaps)
do

```



```

[[ $line =~ ^Private_Clean:\s+(\S+) ]] && ((pkb += ${.sh.match[1]}))
[[ $line =~ ^Private_Dirty:\s+(\S+) ]] && ((pkb += ${.sh.match[1]}))
[[ $line =~ ^Shared_Clean:\s+(\S+) ]] && ((skb += ${.sh.match[1]}))
[[ $line =~ ^Shared_Dirty:\s+(\S+) ]] && ((skb += ${.sh.match[1]}))
[[ $line =~ ^Size:\s+(\S+) ]] && ((szkb += ${.sh.match[1]}))
[[ $line =~ ^Pss:\s+(\S+) ]] && ((psskb += ${.sh.match[1]}))
done

((tkb += pkb))
((tkb += skb))
#((tkb += psskb))

echo "Total private:      $pkb kb"
echo "Total shared:      $skb kb"
echo "Total proc prop:    $psskb kb Pss"
echo "Priv + shared:      $tkb kb"
echo "Size:                $szkb kb"

pmap -d $1 | tail -n 1

```

## References [[edit](#)]

- [Memory usage script](#)

## Various tools [[edit](#)]

- [gdb-heap extension](#)
- [Dr. Memory](#)
- [Pin tool](#)
- [Type-preserving heap profiler](#)
- [↑ {https://gcc.gnu.org/gcc-4.9/changes.html}](https://gcc.gnu.org/gcc-4.9/changes.html)

Category: [Linux Applications Debugging Techniques](#)

This page was last modified on 27 May 2015, at 18:47.

Text is available under the [Creative Commons Attribution-ShareAlike License](#).; additional terms may apply. By using this site, you agree to the [Terms of Use](#) and [Privacy Policy](#).

[Privacy policy](#) [About Wikibooks](#) [Disclaimers](#) [Developers](#) [Cookie statement](#) [Mobile view](#)

