# Linux Applications Debugging Techniques/Print Version

# Linux Applications Debugging Techniques



---

## Preamble

A hands-on guide to debugging applications under Linux, aiming to ease your life as a debugging dog. Applicable to other Unices as well, as long as the tools are available on the target platform.

## Authors

Aurelian Melinte

# Table of Contents



**Not a book title page. Please remove `{{alphabetical}}` from this page.**

# The debugger

## Preparations

A few preparations to ease the debugging trip:

- Have a "symbol server"
- Ship gdbserver with the application for remote debugging
- Embed a breakpoint in the code, at the place of interest, then
  - Start the application
  - Attach to it with the debugger
  - Wait until the breakpoint is hit

## The "symbol server"

One way to easily reach the right code from within the debugger is to build the binaries within an auto-mounted folder, each build in its own sub-folder. The same auto-mount share should be accessible from the machine you are debugging on.

- Export the folder: edit `/etc/exports`
- As root (RedHat): `service autofs start`
- `cd /net/<machine>/path/to/make && make`

## Embedding breakpoints in the source

On x86 platforms:

```
#define EMBEDDED_BREAKPOINT  asm volatile ("int3;")
```

Or a more elaborate one:

```
#define EMBEDDED_BREAKPOINT \
    asm("0:"                              \
        ".pushsection embed-breakpoints;" \
        ".quad 0b;"                       \
        ".popsection;")
```

#### References

- http://mainisusuallyafunction.blogspot.com/2012/01/embedding-gdb-breakpoints-in-c-source.html

## Attaching to a process

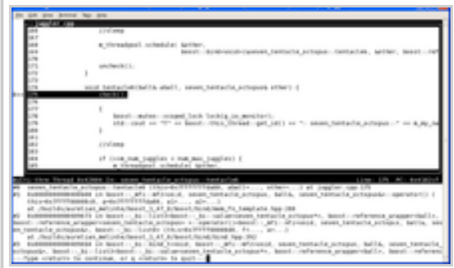
Find out the PID of the process, then:

```
(gdb) attach 1045
Attaching to process 1045
Reading symbols from /usr/lib64/firefox-3.0.18/firefox...(no debugging s
Reading symbols from /lib64/libpthread.so.0...(no debugging symbols four
[Thread debugging using libthread_db enabled]
[New Thread 0x448b4940 (LWP 1063)]
[New Thread 0x428b0940 (LWP 1054)]
....
(gdb) detach
```

## The text user interface

GDB features a text user interface for code, disassembler and registers. For instance:

- **Ctrl-x 1** will show the code pane
- **Ctrl-x a** will hide the TUI panes


GDB TUI

#### References

- http://sourceware.org/gdb/onlinedocs/gdb/TUI.html

## Remote debugging

- On the machine where the application runs (appmachine):
  - If gdbserver is not present , copy it over.
  - Start the application.
  - Start gdbserver: `gdbserver gdbmachine:2345 --attach program`
- On gdbmachine:
  - At the gdb prompt, enter: `target remote appmachine:2345`

Sometimes you may have to tunnel over ssh:

- On gdbmachine:
  - `ssh -L 5432:appmachine:2345 user@appmachine`
  - At the gdb prompt: `target remote localhost:5432`

#### References

- GDB Tunneling (http://www.cucy.net/lacp/archives/000024.html)

## C++ support

Canned gdb macros:

- gdb STL support (http://sourceware.org/gdb/wiki/STLSupport)
- STL macros (and more) (http://www.yolinux.com/TUTORIALS/GDB-Commands.html#STLDEREF)

# The dynamic linker

## Dependencies

- `ldd -d -r /path/to/binary`
- A script to visualize libraries and their dependencies. (http://domseichter.blogspot.com/2008/02/visualize-dependencies-of-binaries-and.html)

## Resolved symbols

To find out which dynamic library is a symbol coming from:

```
$ LD_DEBUG_OUTPUT=sym.log LD_DEBUG=bindings /bin/ls

$ cat sym.log.7688 | grep malloc
      7688:     binding file /lib/i686/cmov/libc.so.6 [0] to /lib/i686/c
      7688:     binding file /lib/i686/cmov/libc.so.6 [0] to /bin/ls [0]
      7688:     binding file /lib/i686/cmov/libc.so.6 [0] to /lib/i686/c
      7688:     binding file /lib/ld-linux.so.2 [0] to /lib/i686/cmov/li
      7688:     binding file /lib/i686/cmov/libc.so.6 [0] to /lib/i686/c
      7688:     binding file /bin/ls [0] to /lib/i686/cmov/libc.so.6 [0]
```

```
$ LD_DEBUG=help /bin/ls
Valid options for the LD_DEBUG environment variable are:

  libs        display library search paths
  reloc       display relocation processing
  files       display progress for input file
  symbols     display symbol table processing
  bindings    display information about symbol binding
  versions    display version dependencies
  all         all previous options combined
  statistics  display relocation statistics
  unused      determined unused DSOs
  help        display this help message and exit

To direct the debugging output into a file instead of standard output
a filename can be specified using the LD_DEBUG_OUTPUT environment variak
```

**References**

- `man 8 ld.so`
- libc symbols visibility & linking (http://www.technovelty.org/linux/libc-symbol-visibility.html)

# Core files

A core dump is a snaphot of the memory of the program, processor registers including program counter and stack pointer and other OS and memory management information, taken at a certain point in time. As such, they are invaluable for capturing the state of rare occurring races and abnormal conditions. One can force a core dump from within the program or from outside at chosen moments. What a core cannot tel is how the application ended up in that state: the core is no replacement for a good log.

## Prerequisites

For a process to be able to dump core, a few prerequisites have to be met:

- the set core size limit should permit it (see the man page for ulimit). E.g.: `ulimit -c unlimited`
- the process to dump core should have write permissions to the folder where the core is to be dumped to (usually the current working directory of the process)

## Where is my core?

Usually the core is dumped in the current working directory of the process. But the OS can be configured otherwise:

```
# cat /proc/sys/kernel/core_pattern
%h-%e-%p.core

# sysctl -w "kernel.core_pattern=/var/cores/%h-%e-%p.core"
```

## Dumping core from outside the program

One possibility is with gdb, if available. This will let the program running:

```
(gdb) attach <pid>
(gdb) generate-core-file <optional-filename>
(gdb) detach
```

Another possibility is to signal the process. This will terminate it, assuming the signal is not caught by a custom signal handler:

```
kill -s SIGABRT <pid>
```

## Dumping core from within the program

Again, there are two possibilities: dump core and terminate the program or dump and continue:

```
void dump_core_and_terminate(void)
{
    abort();
}

void dump_core_and_continue(void)
{
    pid_t child = fork();
    if (child < 0) {
        /*Parent: error*/
    }
    else if (child == 0) {
        dump_core_and_terminate(); /*Child*/
    }
    else {
        /*Parent: continue*/
    }
}
```

## Shared libraries

To obtain a good call stack, it is important that the gdb loads the same libraries that were loaded by the program that generated the core dump. If the machine we are analyzing the core has different libraries (or has them in different places) from the machine the core was dumped, then copy over the libraries to the analyzing machine, in a way that mirrors the dump machine. For instance:

```
$ tree .
.
|-- juggler-29964.core
|-- lib64
|   |-- ld-linux-x86-64.so.2
|   |-- libc.so.6
|   |-- libm.so.6
|   |-- libpthread.so.0
|   `-- librt.so.1
...
```

At the gdb prompt:

```
(gdb) set solib-absolute-prefix ./
(gdb) set solib-search-path .
(gdb) file ../../../../../threadpool/bin.v2/libs/threadpool/example/jugg
Reading symbols from /home/aurelian_melinte/threadpool/threadpool-0_2_5-
```

```
(gdb) core-file juggler-29964.core
Reading symbols from ./lib64/librt.so.1...(no debugging symbols found)..
Loaded symbols for ./lib64/librt.so.1
Reading symbols from ./lib64/libm.so.6...(no debugging symbols found)...
Loaded symbols for ./lib64/libm.so.6
Reading symbols from ./lib64/libpthread.so.0...(no debugging symbols fou
Loaded symbols for ./lib64/libpthread.so.0
Reading symbols from ./lib64/libc.so.6...(no debugging symbols found)...
Loaded symbols for ./lib64/libc.so.6
Reading symbols from ./lib64/ld-linux-x86-64.so.2...(no debugging symbol
Loaded symbols for ./lib64/ld-linux-x86-64.so.2
Core was generated by `../../../../bin.v2/libs/threadpool/example/juggle
Program terminated with signal 6, Aborted.
#0  0x0000003684030265 in raise () from ./lib64/libc.so.6
(gdb) frame 2
#2  0x0000000000404ae1 in dump_core_and_terminate () at juggler.cpp:30
```

## analyze-cores

Here is a script that will generate a basic report per core file. Useful the days when cores are raining on you:

```
#!/bin/bash

#
# A script to extract core-file informations
#


if [ $# -ne 1 ]
then
  echo "Usage: `basename $0` <for-binary-image>"
  exit -1
else
  binimg=$1
fi


# Today and yesterdays cores
cores=`find . -name '*.core' -mtime -1`

#cores=`find . -name '*.core'`


for core in $cores
do
  gdblogfile="$core-gdb.log"
  rm $gdblogfile
```

```
    bininfo=`ls -l $binimg`
    coreinfo=`ls -l $core`

    gdb -batch \
        -ex "set logging file $gdblogfile" \
        -ex "set logging on" \
        -ex "set pagination off" \
        -ex "printf \"**\n** Process info for $binimg - $core \n** Generat
        -ex "printf \"**\n** $bininfo \n** $coreinfo\n**\n\"" \
        -ex "file $binimg" \
        -ex "core-file $core" \
        -ex "bt" \
        -ex "info proc" \
        -ex "printf \"*\n* Libraries \n*\n\"" \
        -ex "info sharedlib" \
        -ex "printf \"*\n* Memory map \n*\n\"" \
        -ex "info target" \
        -ex "printf \"*\n* Registers \n*\n\"" \
        -ex "info registers" \
        -ex "printf \"*\n* Current instructions \n*\n\"" -ex "x/16i \$pc"
        -ex "printf \"*\n* Threads (full) \n*\n\"" \
        -ex "info threads" \
        -ex "bt" \
        -ex "thread apply all bt full" \
        -ex "printf \"*\n* Threads (basic) \n*\n\"" \
        -ex "info threads" \
        -ex "thread apply all bt" \
        -ex "printf \"*\n* Done \n*\n\"" \
        -ex "quit"
done
```

## Canned user-defined commands

Same reporting functionality can be canned for gdb:

```
define procinfo
    printf "**\n** Process Info: \n**\n"
    info proc

    printf "*\n* Libraries \n*\n"
    info sharedlib

    printf "*\n* Memory Map \n*\n"
    info target

    printf "*\n* Registers \n*\n"
    info registers
```

```
    printf "*\n* Current Instructions \n*\n"
    x/16i $pc

    printf "*\n* Threads (basic) \n*\n"
    info threads
    thread apply all bt
end
document procinfo
Infos about the debugee.
end

define analyze
    procinfo

    printf "*\n* Threads (full) \n*\n"
    info threads
    bt
    thread apply all bt full
end
```

### analyze-pid

A script that will generate a basic report and a core file for a running process:

```
#!/bin/bash

#
# A script to generate a core and a status report for a running process.
#


if [ $# -ne 1 ]
then
  echo "Usage: `basename $0` <PID>"
  exit -1
else
  pid=$1
fi


gdblogfile="analyze-$pid.log"
rm $gdblogfile

corefile="core-$pid.core"

gdb -batch \
     -ex "set logging file $gdblogfile" \
```

```
        -ex "set logging on" \
        -ex "set pagination off" \
        -ex "printf \"**\n** Process info for PID=$pid \n** Generated `dat
        -ex "printf \"**\n** Core: $corefile \n**\n\"" \
        -ex "attach $pid" \
        -ex "bt" \
        -ex "info proc" \
        -ex "printf \"*\n* Libraries \n*\n\"" \
        -ex "info sharedlib" \
        -ex "printf \"*\n* Memory map \n*\n\"" \
        -ex "info target" \
        -ex "printf \"*\n* Registers \n*\n\"" \
        -ex "info registers" \
        -ex "printf \"*\n* Current instructions \n*\n\"" -ex "x/16i \$pc"
        -ex "printf \"*\n* Threads (full) \n*\n\"" \
        -ex "info threads" \
        -ex "bt" \
        -ex "thread apply all bt full" \
        -ex "printf \"*\n* Threads (basic) \n*\n\"" \
        -ex "info threads" \
        -ex "thread apply all bt" \
        -ex "printf \"*\n* Done \n*\n\"" \
        -ex "generate-core-file $corefile" \
        -ex "detach" \
        -ex "quit"
```

## Thread Local Storage

TLS data is rather difficult to access with gdb in the core files, and `__tls_get_addr()` cannot be called.

**References**

- __thread variables (http://www.technovelty.org/linux/thread-variable-debug.html)

# The call stack

Sometimes we need the call stack at a certain point in the program. These are the API functions to get basic stack information:

```
#include <execinfo.h>

int backtrace(void **buffer, int size);
char **backtrace_symbols(void *const *buffer, int size);
void backtrace_symbols_fd(void *const *buffer, int size, int fd);

#include <cxxabi.h>
char* __cxa_demangle(const char* __mangled_name, char* __output_buffer,

#include <dlfcn.h>
int dladdr(void *addr, Dl_info *info);
```

Notes:

- C++ symbols are still mangled. Use abi::__cxa_demangle() (http://gcc.gnu.org/onlinedocs/libstdc++/manual/ext_demangling.html) or something similar.
- Some of the these functions do allocate memory - either temporarily either explicitly - and this might be a problem if the program is instable already.
- Some of the these functions do acquire locks (e.g. `dladdr()`).
- Compile with `-rdynamic`
- Link with `-ldl`

To extract more information, use libbfd.

```
class call_stack
{
public:

    static const int depth = 40;
    typedef std::array<void *, depth> stack_t;

    class const_iterator;
    class frame
    {
    public:

        frame(void *addr = 0)
                : _addr(0)
                , _dladdr_ret(false)
                , _binary_name(0)
                , _func_name(0)
                , _demangled_func_name(0)
```

```
                , _delta_sign('+')
                , _delta(0L)
                , _source_file_name(0)
                , _line_number(0)
        {
            resolve(addr);
        }

        // frame(stack_t::iterator& it) : frame(*it) {} //C++0x
        frame(stack_t::const_iterator const& it)
                : _addr(0)
                , _dladdr_ret(false)
                , _binary_name(0)
                , _func_name(0)
                , _demangled_func_name(0)
                , _delta_sign('+')
                , _delta(0L)
                , _source_file_name(0)
                , _line_number(0)
        {
            resolve(*it);
        }

        frame(frame const& other)
        {
            resolve(other._addr);
        }

        frame& operator=(frame const& other)
        {
            if (this != &other) {
                resolve(other._addr);
            }
            return *this;
        }

        ~frame()
        {
            resolve(0);
        }

        std::string to_string() const
        {
            std::ostringstream s;
            s << "[" << std::hex << _addr << "] "
              << demangled_function()
              << " (" << binary_file() << _delta_sign << "0x" << std::he
              << " in " << source_file() << ":" << line_number()
              ;
            return s.str();
```

```
        }

        const void* addr() const               { return _addr; }
        const char* binary_file() const        { return safe(_binary_nam
        const char* function() const           { return safe(_func_name)
        const char* demangled_function() const { return safe(_demangled_
        char        delta_sign() const         { return _delta_sign; }
        long        delta() const              { return _delta; }
        const char* source_file() const        { return safe(_source_fil
        int         line_number() const        { return  _line_number; }

    private:

        const char* safe(const char* p) const { return p ? p : "??"; }

        friend class const_iterator; // To call resolve()
        void resolve(const void * addr)
        {
            if (_addr == addr)
                return;

            _addr = addr;
            _dladdr_ret = false;
            _binary_name = 0;
            _func_name = 0;
            if (_demangled_func_name) {
                free(_demangled_func_name);
                _demangled_func_name = 0;
            }
            _delta_sign = '+';
            _delta = 0L;
            _source_file_name = 0;
            _line_number = 0;

            if (!_addr)
                return;

            _dladdr_ret = (::dladdr(_addr, &_info) != 0);
            if (_dladdr_ret)
            {
                _binary_name = safe(_info.dli_fname);
                _func_name   = safe(_info.dli_sname);
                _delta_sign  = (_addr >=  _info.dli_saddr) ? '+' : '-';
                _delta = ::labs(static_cast<const char *>(_addr) - stati

                 int status = 0;
                 _demangled_func_name = abi::__cxa_demangle(_func_name,
            }
        }
```

```
    private:

        const void* _addr;
        const char* _binary_name;
        const char* _func_name;
        const char* _demangled_func_name;
        char        _delta_sign;
        long        _delta;
        const char* _source_file_name; //TODO: libbfd
        int         _line_number;

        Dl_info     _info;
        bool        _dladdr_ret;
    }; //frame


    class const_iterator
            : public std::iterator< std::bidirectional_iterator_tag
                                  , ptrdiff_t
                                  >
    {
    public:

        const_iterator(stack_t::const_iterator const& it)
                : _it(it)
                , _frame(it)
        {}

        bool operator==(const const_iterator& other) const
        {
            return _frame.addr() == other._frame.addr();
        }

        bool operator!=(const const_iterator& x) const
        {
            return !(*this == x);
        }

        const frame& operator*() const
        {
            return _frame;
        }
        const frame* operator->() const
        {
            return &_frame;
        }

        const_iterator& operator++()
        {
            ++_it;
```

```
            _frame.resolve(*_it);
            return *this;
        }
        const_iterator operator++(int)
        {
            const_iterator tmp = *this;
            ++_it;
            _frame.resolve(*_it);
            return tmp;
        }

        const_iterator& operator--()
        {
            --_it;
            _frame.resolve(*_it);
            return *this;
        }
        const_iterator operator--(int)
        {
            const_iterator tmp = *this;
            --_it;
            _frame.resolve(*_it);
            return tmp;
        }

    private:

        const_iterator();

    private:

        frame                     _frame;
        stack_t::const_iterator  _it;
    }; //const_iterator


    call_stack() : _num_frames(0)
    {
        _num_frames = ::backtrace(_stack.data(), depth);
        assert(_num_frames >= 0 && _num_frames <= depth);
    }

    std::string to_string()
    {
        std::string s;
        const_iterator itEnd = end();
        for (const_iterator it = begin(); it != itEnd; ++it) {
            s += it->to_string();
            s += "\n";
        }
```

```
        return std::move(s);
    }

    virtual ~call_stack()
    {
    }

    const_iterator begin() const { return _stack.cbegin(); }
    const_iterator end() const   { return stack_t::const_iterator(&_stac

private:

    stack_t  _stack;
    int      _num_frames;
};
```

A canned command to resolve a stack address from within gdb:

```
define addrtosym
    if $argc == 1
        printf "[%u]: ", $arg0
        #whatis/ptype EXPR
        #info frame ADDR
        info symbol $arg0
    end
end
document addrtosym
Resolve the address (e.g. of one stack frame). Usage: addrtosym addr0
end
```

# The interposition library

The dynamic liker allows for interception of any function call an application makes to any shared library it uses. As such, interposition is a powerful technique allowing to tune performance, collect runtime statistics, or debug the application without having to instrument the code of that application.

As an example, we can use an interposition library to trace calls, with arguments' values and return codes.

## Call tracing

Note that part of code below is 32-bit x86 and gcc 4.1/4.2 specific.

#### Code intrumentation

In the library, we want to address the following points:

- when a function/method is entered and exited.
- what were the call arguments when the function is entered.
- what was the return code when the function is exited.
- optionally, where was the function called from.

The first one is easy: if requested, the compiler will instrument functions and methods so that when a function/method is entered, a call to an instrumentation function is made and when the function is exited, a similar intrumentation call is made:

```
void __cyg_profile_func_enter(void *func, void *callsite);
void __cyg_profile_func_exit(void *func, void *callsite);
```

This is achieved by compiling the code with the `-finstrument-functions` flag. The above two functions can be used for instance to collect data for coverage; or for profiling. We will use them to print a trace of function calls. Furthermore, we can isolate these two functions and the supporting code in an interposition library of our own. This library can be loaded when and if needed, thus leaving the application code basically unchanged.

Now when the function is entered we can get the arguments of the call:

```
void __cyg_profile_func_enter( void *func, void *callsite )
{
    char buf_func[CTRACE_BUF_LEN+1] = {0};
    char buf_file[CTRACE_BUF_LEN+1] = {0};
    char buf_args[ARG_BUF_LEN + 1] = {0};
    pthread_t self = (pthread_t)0;
    int *frame = NULL;
    int nargs = 0;
```

```
        self = pthread_self();
        frame = (int *)__builtin_frame_address(1); /*of the 'func'*/

        /*Which function*/
        libtrace_resolve (func, buf_func, CTRACE_BUF_LEN, NULL, 0);

        /*From where.  KO with optimizations. */
        libtrace_resolve (callsite, NULL, 0, buf_file, CTRACE_BUF_LEN);

        nargs = nchr(buf_func, ',') + 1; /*Last arg has no comma after*/
        nargs += is_cpp(buf_func);       /*'this'*/
        if (nargs > MAX_ARG_SHOW)
            nargs = MAX_ARG_SHOW;

        printf("T%p: %p %s %s [from %s]\n",
               self, (int*)func, buf_func,
               args(buf_args, ARG_BUF_LEN, nargs, frame),
               buf_file);
    }
```

And when the function is is exited, we get the return value:

```
    void __cyg_profile_func_exit( void *func, void *callsite )
    {
        long ret = 0L;
        char buf_func[CTRACE_BUF_LEN+1] = {0};
        char buf_file[CTRACE_BUF_LEN+1] = {0};
        pthread_t self = (pthread_t)0;

        GET_EBX(ret);
        self = pthread_self();

        /*Which function*/
        libtrace_resolve (func, buf_func, CTRACE_BUF_LEN, NULL, 0);

        printf("T%p: %p %s => %d\n",
               self, (int*)func, buf_func,
               ret);

        SET_EBX(ret);
    }
```

Since these two instrumentation functions are aware of addresses and we actually want the trace to be readable by humans, we need also a way to resolve symbol addresses to symbol names: this is what libtrace_resolve() does.

**Binutils and libbfd**

First, we have to have the symbols information handy. To achieve this, we compile our application with the -g flag. Then, we can map addresses to symbol names and this would normally require writing some code knowledgeable of the ELF format.

Luckily, the there is the binutils package which comes with a library that does just that: libbfd; and with a tool: addr2line. addr2line is a good example on how to use libbfd and I have simply used it to wrap around libbfd. The result is the libtrace_resolve() function.

Since the instrumentation functions are isolated in a stand-alone module, we tell this module the name of the instrumented executable through an environment variable (CTRACE_PROGRAM) that we set before running the program. This is needed to properly init libbfd to search for symbols.

#### Stack layout

To address the first point the work has been architecture-agnostic (actually libbfd is aware of the architecture, but things are hidden behind its API). However, to retrieve function arguments and return values we have to look at the stack, write a bit of architecture-specific code and exploit some gcc quirks. Again, the compilers I have used were gcc 4.1 and 4.2; later or previous versions might work differently. In short:

- x86 dictates that stack grows down.
- GCC dictates how the stack is used - a "typical" stack is depicted below.
- each function has a stack frame marked by the ebp (base pointer) and esp (stack pointer) registers.
- normally, we expect the eax register to contain the reurn code

```
                       \
         +------------+   |
         | arg 2      |   \
         +------------+    >- previous function's stack frame
         | arg 1      |   /
         +------------+   |
         | ret %eip   |   /
         +============+
         | saved %ebp |   \
  %ebp-> +------------+   |
         |            |   |
         |   local    |   \
         | variables, |    >- current function's stack frame
         |    etc.    |   /
         |            |   |
         |            |   |
  %esp-> +------------+   /
```

In an ideal world, the code the compiler generates would make sure that upon instrumenting the exit of a function: the return value is set, then CPU registers pushed on the stack (to ensure the instrumentation function does not affects them), then call the instrumentation function and then pop the registers. This sequence of code would ensure we always get access to the

```
return value in the instrumentation function. The code generated by the compiler is a bit different...
```

Also, in practice, many of gcc's flags affect the stack layout and registers usage. The most obvious ones are:

- -fomit-frame-pointer. This flag affects the stack offset where the arguments are to be found.
- The optimization flags: -Ox; each of these flags aggregates a number of optimizations. These flags did not affected the stack, and, quite amazingly, arguments were always passsed to functions through the stack, regardless of the optimization level. One would have expected that some arguments would pe passed through registers - in which case getting these arguments would have proven to be difficult to impossible. However, these flags did complicated recovering the return code. However, on some architectures, these flags will suck in the -fomit-frame-pointer optimization.

- In any case, be wary: other flags you use to compile your application may reserve surprises.

#### Function arguments

In my tests with the compilers, all arguments were invariably passed through the stack. Hence this is trivial business, affected to a small extent by the -fomit-frame-pointer flag - this flag will change the offset at which arguments start.

How many arguments a function has, how many arguments are on the stack? One way to infer somehow the number of arguments is based on its signature (for C++, beware of the 'this' hidden argument) and this is the technique used in __cyg_profile_func_enter().

Once we know the offset where arguments start on the stack and how many of them there are, we just walk the stack to retrieve their values:

```
char *args(char *buf, int len, int nargs, int *frame)
{
    int i;
    int offset;

    memset(buf, 0, len);

    snprintf(buf, len, "(");
    offset = 1;
    for (i=0; i&lt;nargs && offset&lt;len; i++) {
        offset += snprintf(buf+offset, len-offset, "%d%s",
                         *(frame+ARG_OFFET+i),
                         i==nargs-1 ? " ...)" : ", ");
    }

    return buf;
}
```

#### Function return values

Obtaining the return value proved to be possible only when using the -O0 flag.

Let's look what happens when this method

```
    class B {
        ...
        virtual int m1(int i, int j) {printf("B::m1()\n"); f1(i); return
        ...
    };
```

is instrumented with -O0:

```
      080496a2 <_ZN1B2m1Eii>:
      80496a2:    55                      push   %ebp
      80496a3:    89 e5                   mov    %esp,%ebp
      80496a5:    53                      push   %ebx
      80496a6:    83 ec 24                sub    $0x24,%esp
      80496a9:    8b 45 04                mov    0x4(%ebp),%eax
      80496ac:    89 44 24 04             mov    %eax,0x4(%esp)
      80496b0:    c7 04 24 a2 96 04 08    movl   $0x80496a2,(%esp)
      80496b7:    e8 b0 f4 ff ff          call   8048b6c <__cyg_profile_
      80496bc:    c7 04 24 35 9c 04 08    movl   $0x8049c35,(%esp)
      80496c3:    e8 b4 f4 ff ff          call   8048b7c <puts@plt>
      80496c8:    8b 45 0c                mov    0xc(%ebp),%eax
      80496cb:    89 04 24                mov    %eax,(%esp)
      80496ce:    e8 9d f8 ff ff          call   8048f70 <_Z2f1i>

 ==>  80496d3:    bb 14 00 00 00          mov    $0x14,%ebx
      80496d8:    8b 45 04                mov    0x4(%ebp),%eax
      80496db:    89 44 24 04             mov    %eax,0x4(%esp)
      80496df:    c7 04 24 a2 96 04 08    movl   $0x80496a2,(%esp)
 ==>  80496e6:    e8 81 f5 ff ff          call   8048c6c <__cyg_profile_
      80496eb:    89 5d f8                mov    %ebx,0xfffffff8(%ebp)
 ==>  80496ee:    eb 27                   jmp    8049717 <_ZN1B2m1Eii+0x
      80496f0:    89 45 f4                mov    %eax,0xfffffff4(%ebp)
      80496f3:    8b 5d f4                mov    0xfffffff4(%ebp),%ebx
      80496f6:    8b 45 04                mov    0x4(%ebp),%eax
      80496f9:    89 44 24 04             mov    %eax,0x4(%esp)
      80496fd:    c7 04 24 a2 96 04 08    movl   $0x80496a2,(%esp)
      8049704:    e8 63 f5 ff ff          call   8048c6c <__cyg_profile_
      8049709:    89 5d f4                mov    %ebx,0xfffffff4(%ebp)
      804970c:    8b 45 f4                mov    0xfffffff4(%ebp),%eax
      804970f:    89 04 24                mov    %eax,(%esp)
      8049712:    e8 15 f5 ff ff          call   8048c2c <_Unwind_Resume
```

```
==>  8049717:     8b 45 f8                  mov    0xfffffff8(%ebp),%eax
     804971a:     83 c4 24                  add    $0x24,%esp
     804971d:     5b                        pop    %ebx
     804971e:     5d                        pop    %ebp
     804971f:     c3                        ret
```

Note how the return code is moved into the ebx register - a bit unexpected, since, traditionally, the eax register is used for return codes - and then the instrumentation function is called. Good to retrieve the return value but to avoid that the ebx register gets clobbered in the instrumentation function, we save it upon entering the function and we restore it upon exit.

When the compilation is done with some degree of optimization (-O1...3; shown here is -O2), the code changes:

```
    080498c0 <_ZN1B2m1Eii>:
     80498c0:     55                        push   %ebp
     80498c1:     89 e5                     mov    %esp,%ebp
     80498c3:     53                        push   %ebx
     80498c4:     83 ec 14                  sub    $0x14,%esp
     80498c7:     8b 45 04                  mov    0x4(%ebp),%eax
     80498ca:     c7 04 24 c0 98 04 08      movl   $0x80498c0,(%esp)
     80498d1:     89 44 24 04               mov    %eax,0x4(%esp)
     80498d5:     e8 12 f4 ff ff            call   8048cec <__cyg_profile_
     80498da:     c7 04 24 2d 9c 04 08      movl   $0x8049c2d,(%esp)
     80498e1:     e8 16 f4 ff ff            call   8048cfc <puts@plt>

     80498e6:     8b 45 0c                  mov    0xc(%ebp),%eax
     80498e9:     89 04 24                  mov    %eax,(%esp)
     80498ec:     e8 af f7 ff ff            call   80490a0 <_Z2f1i>
     80498f1:     8b 45 04                  mov    0x4(%ebp),%eax
     80498f4:     c7 04 24 c0 98 04 08      movl   $0x80498c0,(%esp)
     80498fb:     89 44 24 04               mov    %eax,0x4(%esp)
==>  80498ff:     e8 88 f3 ff ff            call   8048c8c <__cyg_profile_
     8049904:     83 c4 14                  add    $0x14,%esp
==>  8049907:     b8 14 00 00 00            mov    $0x14,%eax
     804990c:     5b                        pop    %ebx
     804990d:     5d                        pop    %ebp
==>  804990e:     c3                        ret

     804990f:     89 c3                     mov    %eax,%ebx
     8049911:     8b 45 04                  mov    0x4(%ebp),%eax
     8049914:     c7 04 24 c0 98 04 08      movl   $0x80498c0,(%esp)
     804991b:     89 44 24 04               mov    %eax,0x4(%esp)
     804991f:     e8 68 f3 ff ff            call   8048c8c <__cyg_profile_
     8049924:     89 1c 24                  mov    %ebx,(%esp)
     8049927:     e8 f0 f3 ff ff            call   8048d1c <_Unwind_Resume
```

```
 804992c:	90                   	nop
 804992d:	90                   	nop
 804992e:	90                   	nop
 804992f:	90                   	nop
```

Note how the instrumentation function gets called first and only then the eax register is set with the return value. Thus, if we absolutely want the return code, we are forced to compile with -O0.

**Sample output**

Finally, below are the results. At at shell prompt type:

```
$ export CTRACE_PROGRAM=./cpptraced
$ LD_PRELOAD=./libctrace.so ./cpptraced
```

```
T0xb7c0f6c0: 0x8048d34 main (0 ...) [from ]
./cpptraced: main(argc=1)
T0xb7c0ebb0: 0x80492d8 thread1(void*) (1 ...) [from ]
T0xb7c0ebb0: 0x80498b2 D (134605416 ...) [from cpptraced.cpp:91]
T0xb7c0ebb0: 0x8049630 B (134605416 ...) [from cpptraced.cpp:66]
B::B()
T0xb7c0ebb0: 0x8049630 B => -1209622540 [from ]
D::D(int=-1210829552)
T0xb7c0ebb0: 0x80498b2 D => -1209622540 [from ]
Hello World! It's me, thread #1!
./cpptraced: done.
T0xb7c0f6c0: 0x8048d34 main => -1212090144 [from ]
T0xb740dbb0: 0x8049000 thread2(void*) (2 ...) [from ]
T0xb740dbb0: 0x80498b2 D (134605432 ...) [from cpptraced.cpp:137]
T0xb740dbb0: 0x8049630 B (134605432 ...) [from cpptraced.cpp:66]
B::B()
T0xb740dbb0: 0x8049630 B => -1209622540 [from ]
D::D(int=-1210829568)
T0xb740dbb0: 0x80498b2 D => -1209622540 [from ]
Hello World! It's me, thread #2!
T#2!
T0xb6c0cbb0: 0x8049166 thread3(void*) (3 ...) [from ]
T0xb6c0cbb0: 0x80498b2 D (134613288 ...) [from cpptraced.cpp:157]
T0xb6c0cbb0: 0x8049630 B (134613288 ...) [from cpptraced.cpp:66]
B::B()
T0xb6c0cbb0: 0x8049630 B => -1209622540 [from ]
D::D(int=0)
T0xb6c0cbb0: 0x80498b2 D => -1209622540 [from ]
Hello World! It's me, thread #3!
T#1!
T0xb7c0ebb0: 0x80490dc wrap_strerror_r (134525680 ...) [from cpptraced.cpp:105]
T0xb7c0ebb0: 0x80490dc wrap_strerror_r => -1210887643 [from ]
T#1+M2 (Success)
T0xb740dbb0: 0x80495a0 D::m1(int, int) (134605432, 3, 4 ...) [from cpptraced.cpp:141]
D::m1()
T0xb740dbb0: 0x8049522 B::m2(int) (134605432, 14 ...) [from cpptraced.cpp:69]
B::m2()
T0xb740dbb0: 0x8048f70 f1 (14 ...) [from cpptraced.cpp:55]
f1 14
T0xb740dbb0: 0x8048ee0 f2(int) (74 ...) [from cpptraced.cpp:44]
f2 74
T0xb740dbb0: 0x8048e5e f3 (144 ...) [from cpptraced.cpp:36]
f3 144
T0xb740dbb0: 0x8048e5e f3 => 80 [from ]
```

```
T0xb740dbb0: 0x8048ee0 f2(int) => 70 [from ]
T0xb740dbb0: 0x8048f70 f1 => 60 [from ]
T0xb740dbb0: 0x8049522 B::m2(int) => 21 [from ]
T0xb740dbb0: 0x80495a0 D::m1(int, int) => 30 [from ]
T#2!
T#3!
```

Note how libbfd fails to resolve some addresses when the function gets inlined.

**Resources**

- Code (http://freeshell.de/~amelinte/software.html)
- Overview of GCC on x86 platforms (http://pdos.csail.mit.edu/6.828/2004/lec/l2.html)
- The Intel stack (http://dsrg.mff.cuni.cz/~ceres/sch/osy/text/ch03s02s02.php)

# Memory issues

Linux Applications Debugging Techniques/Memory issues

# Leaks

## What to look for

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

Valgrind should be the first stop for any memory related issue. However:

- it slows down the program by at least one order of magnitude, in particular C++ programs.
- from experience, some versions might have difficulties tracking `mmap()` allocated memory.
- on amd64, the vex dissassembler is likely to fail (v3.7)
- 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/valgrin
    -v \
    --error-limit=no \
    --num-callers=40 \
    --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

- http://gcc.gnu.org/wiki/Mudflap_Pointer_Debugging

## mtrace

The GNU C library comes with a built-in functionality to help detecting memory issues: `mtrace()`.

#### The basics

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>
...
      mtrace();
...
      std::string* pstr = new std::string("leak");
...
      char *leak = (char*)malloc(1024);
...
      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/memtra
```

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 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/
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
...
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   <== mal
#2  0xb7e95c01 in std::string::_Rep::_S_create () from /usr/lib/libstdc+
#3  0xb7e96f05 in ?? () from /usr/lib/libstdc++.so.6
#4  0xb7e970b7 in std::basic_string<char, std::char_traits<char>, std::a
```

```
#5  0x08048a58 in main () at main.cpp:25           <== new std::string('
...

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

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

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/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()`).

#### A couple of caveats

- 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 
```

- 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.

#### What we got

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/memtra
```

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    <==
**[ 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) <==
[0x80489cf] __gxx_personality_v0 (./dleaker+27f)
[0x40178450] __libc_start_main (/lib/i686/cmov/libc.so.6+e0)        <==
[0x8048791] __gxx_personality_v0 (./dleaker+41)
**] Stack
```

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

#### File and line

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.
- The ultimate solution would be to use `libbfd` (`binutils`). An alternative to `libbfd` could be to use `libcwd` instead.

The third solution could look something like:

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

#include <execinfo.h>
#include <signal.h>
#include <bfd.h>
#include <unistd.h>

/* globals retained across calls to resolve. */
static bfd* abfd = 0;
static asymbol **syms = 0;
static asection *text = 0;

static void resolve(char *address) {
    if (!abfd) {
        char ename[1024];
        int l = readlink("/proc/self/exe",ename,sizeof(ename));
        if (l == -1) {
          perror("failed to find executable\n");
          return;
        }
        ename[l] = 0;

        bfd_init();

        abfd = bfd_openr(ename, 0);
        if (!abfd) {
            perror("bfd_openr failed: ");
            return;
        }

        /* oddly, this is required for it to work... */
```

```
            bfd_check_format(abfd,bfd_object);

            unsigned storage_needed = bfd_get_symtab_upper_bound(abfd);
            syms = (asymbol **) malloc(storage_needed);
            unsigned cSymbols = bfd_canonicalize_symtab(abfd, syms);

            text = bfd_get_section_by_name(abfd, ".text");
    }

    long offset = ((long)address) - text->vma;
    if (offset > 0) {
        const char *file;
        const char *func;
        unsigned line;
        if (bfd_find_nearest_line(abfd, text, syms, offset, &file, &func
            printf("file: %s, line: %u, func %s\n",file,line,func);
    }
}
```

The downside is that it takes a quite heavy toll on the performance of the program.

#### Resources

- The GNU C library manual (http://www.gnu.org/s/hello/manual/libc/Allocation-Debugging.html)
- Using libbfd (http://www.beowulf.org/archive/2007-June/018558.html)
- Linux Programming Toolkit (http://freeshell.de/~amelinte/software.html)

## mallinfo

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() << s
```

```
        assert(second > first);

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

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

    return 0;
}
```

**References**

- mallinfo (http://www.gnu.org/software/libc/manual/html_node/Statistics-of-Malloc.html)
- mallinfo deprecated (http://udrepper.livejournal.com/20948.html)

## /proc

Coarse grained information can be obtained from /proc:

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

IFS=$'\n'

for line in $(</proc/$1/smaps)
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**


- Memory usage script (http://permalink.gmane.org/gmane.comp.video.gstreamer.devel/10609)

# Heap corruption

## Electric Fence

Electric Fence is still the reference for dealing with heap corruption, even if not maintined for a while. RedHat ships a version that can be used as an interposition library.

Drawback: might not work with code that uses `mmap()` to allocate memory.

## Duma

Duma is a fork of Electric Fence.

## glibc builtin

`man (3) malloc`: Recent versions of Linux libc (later than 5.4.23) and GNU libc (2.x) include a malloc implementation which is tunable via environment variables. When `MALLOC_CHECK_` is set, a special (less efficient) implementation is used which is designed to be tolerant against simple errors, such as double calls of free() with the same argument, or overruns of a single byte (off-by-one bugs). Not all such errors can be protected against, however, and memory leaks can result. If `MALLOC_CHECK_` is set to 0, any detected heap corruption is silently ignored and an error message is not generated; if set to 1, the error message is printed on stderr, but the program is not aborted; if set to 2, abort() is called immediately, but the error message is not generated; if set to 3, the error message is printed on stderr and program is aborted. This can be useful because otherwise a crash may happen much later, and the true cause for the problem is then very hard to track down.

# Stack corruption

Stack corruption is rather hard to diagnose. Luckily, gcc 4.x can instrument the code to check for stack corruption:

- **-fstack-protector**
- **-fstack-protector-all**

gcc will add guard variables and code to check for buffer overflows upon exiting a function. A quick example:

```
/* Compile with: gcc -ggdb -fstack-protector-all stacktest.c */

#include <stdio.h>
#include <string.h>

void bar(char* str)
{
    char buf[4];
    strcpy(buf, str);
}

void foo()
{
    printf("It survived!");
}

int main(void)
{
    bar("Longer than 4.");
    foo();
    return 0;
}
```

When run, the program will dump core:

```
$ ./a.out
*** stack smashing detected ***: ./a.out terminated
Aborted (core dumped)
```

```
Core was generated by `./a.out'.
Program terminated with signal 6, Aborted.
#0  0x0000003684030265 in raise () from /lib64/libc.so.6
(gdb) bt full
#0  0x0000003684030265 in raise () from /lib64/libc.so.6
No symbol table info available.
```

```
#1  0x0000003684031d10 in abort () from /lib64/libc.so.6
No symbol table info available.
#2  0x000000368406a84b in __libc_message () from /lib64/libc.so.6
No symbol table info available.
#3  0x00000036840e8ebf in __stack_chk_fail () from /lib64/libc.so.6
No symbol table info available.
#4  0x0000000000400584 in bar (str=0x400715 "Longer than 4.") at stackte
        buf =           "Long"
#5  0x00000000004005e3 in main () at stacktest.c:19
No locals.
```

# Deadlocks

## Analysis

Searching for a deadlock means reconstructing the graph of dependencies between threads and resources (mutexes, semaphores, condition variables, etc.) - who owns what and who wants to acquire what. A typical deadlock would look like a loop in that graph. The task is tedious, as some of the parameters we are looking for have been optimized by the compiler into registers.

Below is an analysis of an x86_64 deadlock. On this platform, register r8 is the one containing the first argument: the address of the mutex:

```
(gdb) thread apply all bt
...
Thread 4 (Thread 0x419bc940 (LWP 12275)):
#0  0x0000003684c0d4c4 in __lll_lock_wait () from /lib64/libpthread.so.6
#1  0x0000003684c08e1a in _L_lock_1034 () from /lib64/libpthread.so.0
#2  0x0000003684c08cdc in pthread_mutex_lock () from /lib64/libpthread.s
#3  0x0000000000400a50 in thread1 (threadid=0x1) at deadlock.c:66
#4  0x0000003684c0673d in start_thread () from /lib64/libpthread.so.0
#5  0x00000036840d3d1d in clone () from /lib64/libc.so.6

Thread 3 (Thread 0x421bd940 (LWP 12276)):
#0  0x0000003684c0d4c4 in __lll_lock_wait () from /lib64/libpthread.so.6
#1  0x0000003684c08e1a in _L_lock_1034 () from /lib64/libpthread.so.0
#2  0x0000003684c08cdc in pthread_mutex_lock () from /lib64/libpthread.s
#3  0x0000000000400c07 in thread2 (threadid=0x2) at deadlock.c:111
#4  0x0000003684c0673d in start_thread () from /lib64/libpthread.so.0
#5  0x00000036840d3d1d in clone () from /lib64/libc.so.6
...


(gdb) thread 4
[Switching to thread 4 (Thread 0x419bc940 (LWP 12275))]#2  0x0000003684c
   from /lib64/libpthread.so.0

(gdb) frame 2
#2  0x0000003684c08cdc in pthread_mutex_lock () from /lib64/libpthread.s

(gdb) info reg
...
r8             0x6015a0 6296992
...

(gdb) p *(pthread_mutex_t*)0x6015a0
$3 = {
  __data = {
```

```
      __lock = 2,
      __count = 0,
      __owner = 12276,   <== T3
      __nusers = 1,
      __kind = 0,        <== non-recursive
      __spins = 0,
      __list = {
        __prev = 0x0,
        __next = 0x0
      }
    },
    __size =     "\002\000\000\000\000\000\000\000\364/\000\000\001", '\00
    __align = 2
  }

  (gdb) thread 3
  [Switching to thread 3 (Thread 0x421bd940 (LWP 12276))]#0  0x0000003684c
     from /lib64/libpthread.so.0

  (gdb) bt
  #0  0x0000003684c0d4c4 in __lll_lock_wait () from /lib64/libpthread.so.0
  #1  0x0000003684c08e1a in _L_lock_1034 () from /lib64/libpthread.so.0
  #2  0x0000003684c08cdc in pthread_mutex_lock () from /lib64/libpthread.s
  #3  0x0000000000400c07 in thread2 (threadid=0x2) at deadlock.c:111
  #4  0x0000003684c0673d in start_thread () from /lib64/libpthread.so.0
  #5  0x00000036840d3d1d in clone () from /lib64/libc.so.6
  #2  0x0000003684c08cdc in pthread_mutex_lock () from /lib64/libpthread.s

  (gdb) info reg
  ...
  r8             0x6015e0 6297056
  ...

  (gdb) p *(pthread_mutex_t*)0x6015e0
  $4 = {
    __data = {
      __lock = 2,
      __count = 0,
      __owner = 12275,  <=== T4
      __nusers = 1,
      __kind = 0,
      __spins = 0,
      __list = {
        __prev = 0x0,
        __next = 0x0
      }
    },
    __size =     "\002\000\000\000\000\000\000\000\363/\000\000\001", '\00
    __align = 2
  }
```

Threads 3 and 4 are deadlocking over two mutexes.

Note: If gdb is unable to find the symbol pthread_mutex_t because it has not loaded the symbol table for pthreadtypes.h, you can still print the individual members of the struct as follows:

```
(gdb) print *((int*)(0x6015e0))
$4 = 2
(gdb) print *((int*)(0x6015e0)+1)
$5 = 0
(gdb) print *((int*)(0x6015e0)+2)
$6 = 12275
```

## Automation

An interposition library can be built to automate deadlock analysis (http://linuxgazette.net/150/melinte.html) .

# Race conditions

## Valgrind Helgrind (http://valgrind.org/docs/manual/hg-manual.html)

- [v 3.7] On amd64 platforms it does not survive for long because of the vex disassembler.

## Valgrind Drd (http://valgrind.org/docs/manual/drd-manual.html)

- Same.

## Relacy (http://www.1024cores.net/home/relacy-race-detector)

- C++0x/11 synchronization modeler/unit tests tool.

## Promela (http://en.wikipedia.org/wiki/Promela)

# Resource leaks

## Zombie threads

Any thread that has terminated but has not been joined or detached will leak OS resources until the process terminates. Unfortunately, neither `/proc` nor `gdb` will show you these zombie threads, at least not on some kernels.

One way to get them is with a gdb canned command:

```
#
#
#
define trace_call
    b $arg0
        commands
        bt full
        continue
        end
end
document trace_call
Trace specified call with call stack to screen. Example:
    set breakpoint pending on
        set pagination off
        set logging on
    trace_call __pthread_create_2_1
end
```

```
Using host libthread_db library "/lib/i686/cmov/libthread_db.so.1".
(gdb) trace_call __pthread_create_2_1
Function "__pthread_create_2_1" not defined.
Breakpoint 1 (__pthread_create_2_1) pending.
(gdb) trace_call __pthread_create_2_0
Function "__pthread_create_2_0" not defined.
Breakpoint 2 (__pthread_create_2_0) pending.
(gdb) r
Starting program: /home/amelinte/projects/articole/wikibooks/debug/plock
[Thread debugging using libthread_db enabled]
Breakpoint 3 at 0xb7f9b746
Pending breakpoint "__pthread_create_2_1" resolved
Breakpoint 4 at 0xb7f9c395
Pending breakpoint "__pthread_create_2_0" resolved
[New Thread 0xb7e48ad0 (LWP 8635)]
[Switching to Thread 0xb7e48ad0 (LWP 8635)]

Breakpoint 3, 0xb7f9b746 in pthread_create@@GLIBC_2.1 () from /lib/i686/
#0  0xb7f9b746 in pthread_create@@GLIBC_2.1 () from /lib/i686/cmov/libpt
```

```
No symbol table info available.
#1  0x08048a7f in main (argc=4, argv=0xbfceb714) at plock.c:97
        s = 0
        tnum = 0
        opt = -1
        num_threads = 3
        tinfo = (struct thread_info *) 0x833b008
        attr = {__size = '\0' <repeats 13 times>, "\020", '\0' <repeats
        stack_size = -1
        res = (void *) 0x0
[New Thread 0xb7e47b90 (LWP 8638)]
Thread 1: top of stack near 0xb7e473c8; argv_string=foo
```

Another way is to use (again) an interposition library:

```
/*
 *  Hook library. Usage:
 *    gcc -c -g -Wall -fPIC libhook.c -o libhook.o
 *    ld -o libhook.so libhook.o -shared -ldl
 *    LD_PRELOAD=./libhook.so program arguments
 *
 *  Copyright 2012 Aurelian Melinte.
 *  Released under GPL 3.0 or later.
 */

#define _GNU_SOURCE
#include <dlfcn.h>

#include <signal.h>
#include <execinfo.h>

#include <errno.h>
#include <stdlib.h>
#include <stdio.h>  /*printf*/
#include <unistd.h>

#include <pthread.h>

#include <assert.h>



typedef int (*lp_pthread_mutex_func)(pthread_mutex_t *mutex);
typedef int (*pthread_create_func)(pthread_t *thread,
                                   const pthread_attr_t *attr,
                                                                void
static pthread_create_func  _pthread_create_hook = NULL;

```

```
static int
hook_one(pthread_create_func *fptr, const char *fname)
{
    char *msg = NULL;

    assert(fname != NULL);

    if (*fptr == NULL) {
        printf("dlsym : wrapping %s\n", fname);
        *fptr = dlsym(RTLD_NEXT, fname);
        printf("next_%s = %p\n", fname, *fptr);
        if ((*fptr == NULL) || ((msg = dlerror()) != NULL)) {
            printf("dlsym %s failed : %s\n", fname, msg);
            return -1;
        } else {
            printf("dlsym: wrapping %s done\n", fname);
            return 0;
        }
    } else {
        return 0;
    }
}


static void
hook_funcs(void)
{
    if (_pthread_create_hook == NULL) {
        int rc = hook_one(&_pthread_create_hook, "pthread_create");
        if (NULL == _pthread_create_hook || rc != 0) {
            printf("Failed to hook.\n");
            exit(EXIT_FAILURE);
        }
    }
}


/*
 *
 */


int
pthread_create(pthread_t *thread,
               const pthread_attr_t *attr,
               void *(*start_routine) (void *), void *arg)
{
#define SIZE 40
    void *buffer[SIZE] = {0};
```

```
        int nptrs = 0;

    int rc = EINVAL;

        rc = _pthread_create_hook(thread, attr, start_routine, arg);

    printf("*** pthread_create:\n");
    nptrs = backtrace(buffer, SIZE);
    backtrace_symbols_fd(buffer, nptrs, STDOUT_FILENO);

    return rc;
}

/*
 *
 */

void _init()  __attribute__((constructor));
void
_init()
{
    printf("*** _init().\n");
    hook_funcs();
}


void  _fini()  __attribute__((destructor));
void
_fini()
{
    printf("*** _fini().\n");
}
```

The output is a bit rough but it can be refined down to file and line by replacing `backtrace_symbols_fd()` with appropriate code:

```
*** pthread_create:
./libhook.so(pthread_create+0x8c)[0x400215d3]
./plock[0x8048a7f]
/lib/i686/cmov/libc.so.6(__libc_start_main+0xe0)[0x4006f450]
./plock[0x8048791]
```

## File descriptors

As just about anything is a file (folders, sockets, pipes, etc, etc...), just about anything can result in a file descriptor that needs to be closed. `/proc` can help:

```
# tree /proc/26041
/proc/26041
...
|-- fd                    # Open files descriptors
|   |-- 0 -> /dev/pts/21
|   |-- 1 -> /dev/pts/21
|   |-- 2 -> /dev/pts/21
|   `-- 3 -> socket:[113497835]
|-- fdinfo
|   |-- 0
|   |-- 1
|   |-- 2
|   `-- 3
...
```

The `trace_call` command for `gdb` can help with the call stack.

If gdb is not available on the machine, an interposition library hooking `open()`, `pipe()`, `socket()`, etc. can be built.

Other tools that can be used:

- lsof
- fuser

## Ports

Which process is using a port? As root:

```
# netstat -tlnp
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address               Foreign Address
tcp        0      0 0.0.0.0:36510               0.0.0.0:*
tcp        0      0 127.0.0.1:2207              0.0.0.0:*
...
```

```
# lsof
COMMAND     PID             USER   FD      TYPE             DEVICE
init          1             root  cwd       DIR              253,0
...
python     3438             root    4u     IPv4              11416

# lsof -i :2207
COMMAND   PID USER   FD   TYPE DEVICE SIZE NODE NAME
python  3438 root    4u  IPv4  11416       TCP localhost.localdomain:220
```

Other tools:

- fuser

## IPC

For semaphores, shared memory and message queues.

- ipcs
- ipcrm

```
# ipcs -spt
------ Semaphore Operation/Change Times --------
semid    owner      last-op                    last-changed
187826177 aurelian_m  Fri Feb 10 09:37:26 2012   Fri Feb 10 09:33:39 201
187858946 aurelian_m  Fri Feb 10 09:52:11 2012   Fri Feb 10 09:50:44 201
```

# Aiming for and measuring performance

## gprof & -pg

To profile the application with gprof:

- Compile the code with `-pg`
- Link with `-pg`
- Run the application. This creates a file `gmon.out` in the current folder of the application.
- At the prompt, in the folder where gmon.out lives: `gprof path-to-application`

## PAPI

The Performance Application Programming Interface (PAPI) (http://icl.cs.utk.edu/papi/) offers the programmer access to the performance counter hardware found in most major microprocessors. With a decent C++ wrapper (http://freeshell.de/~amelinte /software.html) , measuring branch mispredictions and cache misses (and much more) is literally one line of code away.

As an example, lets look a bit at these lines:

```
    const int nlines = 196608;
    const int ncols  = 64;
    char ctrash[nlines][ncols];
    {
        int x;
        papi::counters<papi::stdout_print> pc("by column"); //<== the fam
        for (int c = 0; c < ncols; ++c) {
            for (int l = 0; l < nlines; ++l) {
                x = ctrash[l][c];
            }
        }
    }
```

The code just loops over an array but in the wrong order: the innermost loop iterates on the outer index. While the result is the same whether we loop over the first index first or over the last one, theorically, to preserve cache locality, the innermost loop should iterate over the innermost index. This should make a big difference for the time it takes to iterate over the array:

```
    {
        int x;
        papi::counters<papi::stdout_print> pc("By line");
        for (int l = 0; l < nlines; ++l) {
            for (int c = 0; c < ncols; ++c) {
                x = ctrash[l][c];
```

```
            }
        }
    }
```

**papi::counters** is a class wrapping around PAPI functionality. It will take a snaphost of some performance counters (in our case, we are interested in cache misses and in branch mispredictions) when a counters object is instantiated and another snapshot when the object is destroyed. Then it will print out the differences.

A first measure, with non-optimized code (-O0), shows the following:

Delta by column:

```
PAPI_TOT_INS (Total instructions): 188744788 (380506167-191761379)
PAPI_TOT_CYC (Total cpu cycles): 92390347 (187804288-95413941)
PAPI_L1_DCM (L1 load  misses): 28427 (30620-2193)                    <==
PAPI_L2_DCM (L2 load  misses): 102 (1269-1167)
PAPI_BR_MSP (Branch mispredictions): 176 (207651-207475)             <==
```

Delta By line:

```
PAPI_TOT_INS (Total instructions): 190909841 (191734047-824206)
PAPI_TOT_CYC (Total cpu cycles): 94460862 (95387664-926802)
PAPI_L1_DCM (L1 load  misses): 403 (2046-1643)                       <==
PAPI_L2_DCM (L2 load  misses): 21 (1081-1060)
PAPI_BR_MSP (Branch mispredictions): 205934 (207350-1416)            <==
```

While the cache misses have indeed improved, branch mispredictions exploded. Not exactly a good tradeoff. Down in the pipeline of the processor, a comparison operation translates into a branch operation. Something is funny with the unoptimized code the compiler generated.

Maybe the optimized code (-O2) is behaving better? Or maybe not:

Delta by column:

```
PAPI_TOT_INS (Total instructions): 329 (229368-229039)
PAPI_TOT_CYC (Total cpu cycles): 513 (186217-185704)
PAPI_L1_DCM (L1 load  misses): 2 (1523-1521)
PAPI_L2_DCM (L2 load  misses): 0 (993-993)
PAPI_BR_MSP (Branch mispredictions): 7 (1287-1280)
```

Delta By line:

```
PAPI_TOT_INS (Total instructions): 330 (209614-209284)
PAPI_TOT_CYC (Total cpu cycles): 499 (173487-172988)
PAPI_L1_DCM (L1 load  misses): 2 (1498-1496)
PAPI_L2_DCM (L2 load  misses): 0 (992-992)
PAPI_BR_MSP (Branch mispredictions): 7 (1225-1218)
```

This time the compiler optimized the loops out! It figured we do not really use the data in the array, so it got rid of. Completely!

Let's see how this code behaves:

```
    {
        int x;
        papi::counters<papi::stdout_print> pc("by column");
        for (int c = 0; c < ncols; ++c) {
            for (int l = 0; l < nlines; ++l) {
                x = ctrash[l][c];
                ctrash[l][c] = x + 1;
            }
        }
    }
```

Delta by column:

```
PAPI_TOT_INS (Total instructions): 62918492 (63167552-249060)
PAPI_TOT_CYC (Total cpu cycles): 224705473 (224904307-198834)
PAPI_L1_DCM (L1 load  misses): 12415661 (12417203-1542)
PAPI_L2_DCM (L2 load  misses): 9654638 (9655632-994)
PAPI_BR_MSP (Branch mispredictions): 14217 (15558-1341)
```

Delta By line:

```
PAPI_TOT_INS (Total instructions): 51904854 (115092642-63187788)
PAPI_TOT_CYC (Total cpu cycles): 25914254 (250864272-224950018)
PAPI_L1_DCM (L1 load  misses): 197104 (12614449-12417345)
PAPI_L2_DCM (L2 load  misses): 6330 (9662090-9655760)
PAPI_BR_MSP (Branch mispredictions): 296 (16066-15770)
```

Both cache misses and branch mispredictions improved by at least an order of magnitude. A run with unoptimized code will show the same order of improvement.

**References**

- Locality of reference

## OProfile

OProfile offers access to the same hardware counters as PAPI but without having to instrument the code:

- It is coarser grained than PAPI - at function level.
- Some out of the box kernels (RedHat) are not OProfile-friendly.
- You need root access.

```
#!/bin/bash

#
# A script to OProfile a program.
# Must be run as root.
#


if [ $# -ne 1 ]
then
  echo "Usage: `basename $0` <for-binary-image>"
  exit -1
else
  binimg=$1
fi

opcontrol --stop
opcontrol --shutdown

# Out of the box RedHat kernels are OProfile repellent.
opcontrol --no-vmlinux
opcontrol --reset

# List of events for platform to be found in /usr/share/oprofile/<>/ever
opcontrol --event=L2_CACHE_MISSES:1000


opcontrol --start

$binimg

opcontrol --stop
opcontrol --dump


rm $binimg.opreport.log
opreport > $binimg.opreport.log

rm $binimg.opreport.sym
opreport -l > $binimg.opreport.sym


opcontrol --shutdown
opcontrol --deinit
echo "Done"
```

**References**

- OP Manual (http://oprofile.sourceforge.net/doc/index.html)
- IBM OP Intro (http://www.ibm.com/developerworks/systems/library/es-oprofile/)

# Appendices

## Things to watch for

#### Interrupted calls

A number of API functions return an error code if the call was interrupted by a signal. Usually this is not an error by itself and the call should be restarted. For instance:

```
int raccept( int s, struct sockaddr *addr, socklen_t *addrlen )
{
     int rc;

     do {
         rc = accept( s, addr, addrlen );
     } while ( rc == -1 && errno == EINTR );

     return rc;
}
```

The list of interruptible function differs from Unix-like platform to platform. For Linux see signal(7) (http://www.kernel.org/doc/man-pages/online/pages/man7/signal.7.html) .

#### Spurious wake-ups

Threads waiting on a pthreads condition variable can be waken up even if the condition hs not been met. Upon waking up, the condition should be explicitly checked and return waiting if it is not met.

## /proc

A pseudo-filesystem exposing information about running processes:

```
# tree /proc/26041
/proc/26041
...
|-- cmdline              # Command line
|-- cwd ->               /current/working/folder/for/PID
|-- environ              # Program environment variables
|-- exe -> /bin/su
|-- fd                   # Open files descriptors
|   |-- 0 -> /dev/pts/21
|   |-- 1 -> /dev/pts/21
|   |-- 2 -> /dev/pts/21
|   `-- 3 -> socket:[113497835]
|-- fdinfo
```

```
|   |-- 0
|   |-- 1
|   |-- 2
|   `-- 3
|-- latency
|-- limits
|-- maps
|-- mem
|-- mountinfo
|-- mounts
|-- mountstats
...

# cat /proc/26041/status
...
VmPeak:   103276 kB     # Max virtual memory reached
VmSize:   103196 kB     # Current VM
VmLck:         0 kB
VmHWM:      1492 kB
VmRSS:      1488 kB     # Live memory used
...
Threads:        1
...
```

#### References

- Man page (http://www.kernel.org/doc/man-pages/online/pages/man5/proc.5.html)
- Kernel doc (http://www.kernel.org/doc/Documentation/filesystems/proc.txt)

#### sysstat, sar

- Site (http://sebastien.godard.pagesperso-orange.fr/)

## Other tools

#### addr2line

Given an address in an executable or an offset in a section of a relocatable object, `addr2line` translates it into file name and line number.

#### c++filt

A tool to demangle symbol names.

#### objdump

- Disassemble binary, with source code: `objdump -C -S -r -R -l <binary>`

# References and further reading

## Books & articles


- Agar, Eric; Writing Reliable AIX Daemons (http://www.redbooks.ibm.com/redbooks/pdfs/sg244946.pdf)
- McKenney, Paul; Is Parallel Programming Hard (http://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html) (git (git://git.kernel.org/pub/scm/linux/kernel/git/paulmck/perfbook.git%20) , blog (http://paulmck.livejournal.com/) , other papers (http://www.rdrop.com/users/paulmck/) )


## Software

- Linux Programming Tools (http://freeshell.de/~amelinte/software.html)

# GNU Free Documentation License



## 0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

## 1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

# 2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

# 3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

# 4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
D. Preserve all the copyright notices of the Document.
E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
H. Include an unaltered copy of this License.
I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified version.
N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

# 5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

# 6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

# 7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

# 8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

# 9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

# 10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

# 11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

# How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright (c) YEAR YOUR NAME.
> Permission is granted to copy, distribute and/or modify this document
> under the terms of the GNU Free Documentation License, Version 1.3
> or any later version published by the Free Software Foundation;
> with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.
> A copy of the license is included in the section entitled "GNU
> Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the
> Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.