Mudflap: Pointer Use Checking for C/C++

Frank Ch. Eigler
Red Hat
`fche@redhat.com`

Abstract

Mudflap is a pointer use checking technology based on compile-time instrumentation. It transparently adds protective code to a variety of potentially unsafe C/C++ constructs that detect actual erroneous uses at run time. The class of errors detected includes the most common and annoying types: NULL pointer dereferencing, running off the ends of buffers and strings, leaking memory. Mudflap has heuristics that allow some degree of checking even if only a subset of a program's object modules are instrumented.

1 Motivation

C, and to a lesser extent C++, are sometimes jovially referred to as a "portable assembly language." This means that they are portable across platforms, but are low level enough to comfortably deal with hardware and raw bits in memory. This makes them particularly suited for writing systems software such as operating systems, databases, network servers, and data/language processors. These types of software are notorious for pointer-based data structures and algorithms, which C/C++ make easy to express. However, the runtime model of C/C++ does not include any checking of pointer use, so errors can easily creep in.

Several kinds of pointer use errors are widely known by every C/C++ programmer. Accessing freed objects, going past buffer boundaries, dereferencing `NULL` or other bad pointers, can each result in a spectrum of effects, from nothing, through random glitches and outright crashes, to security breaches. Such bugs can induce hard-to-debug delayed failures. Many recent security vulnerabilities of operating systems result from simple stack-smashing *buffer overrun* errors, where pointers go beyond their bounds to corrupt memory, under the influence of malevolent input.

There exist several technologies for catching pointer use errors. They have distinct approaches and capability/performance tradeoffs. For example, from a debugging point of view, it is better to catch the memory corruption at the moment it occurs, because context will be fresh and available. On the other hand, for security protection of a deployed program, it may be enough to catch an error just in time to prevent a breach, which might be much later.

A large class of pointer use errors relates to heap allocation. Writing past the end of a heap object, or accessing a pointer after a `free` can sometimes be detected with nothing more than a library that replaces the standard library's heap functions (`malloc`, `free`, etc.). The Electric Fence[1] package, for example, can manage heap objects that carefully abut inac-

[1] `ftp://ftp.perens.com/pub/ElectricFence/`

cessible virtual memory pages. A buffer overrun there causes an instant segmentation fault. Some libraries provide a protected padding area around buffers. This padding is filled with code that can be periodically checked for changes, so program errors can be detected at a coarser granularity.

The bounded-pointers GCC extension[2] addresses pointer errors by replacing simple pointers with a three-word struct that also contains the legal bounds for that pointer. This changes the system ABI, making it necessary to recompile the entire application. The bounds are computed upon assignment from the address-of operator, and constructed for system calls within an instrumented version of the standard library. Each use of the pointer is quickly checked against its own bounds, and the application aborts upon a violation. Because there is no database of live objects, an instrumented program can offer no extra information to help debug the problem.

The gcc-checker extension[3] addresses pointer errors by mapping all pointer manipulation operations, and all variable lifetime scopes, to calls into a runtime library. In this scheme, the instrumentation is heavy-weight, but the pointer representation in memory remains ABI-compatible. It may be possible to detect the moment a pointer becomes invalid (say, through a bad assignment or increment), before it is ever used to access memory.

The StackGuard GCC extension[4] addresses stack smashing attacks via buffer overruns. It does this by instrumenting a function to rearrange its stack frame, so that arrays are placed away from vulnerable values like return addresses. Further guard padding is added around arrays and is checked before a function returns. This is light-weight and reasonably effective, but provides no general protection for pointer errors or debugging assistance.

The valgrind package[5] is a simulation-based tool for detecting a broad class of pointer use errors. It contains a virtual machine that tracks processor operations, including memory loads and stores and even register arithmetic, to check operations for validity. While it works on unmodified executables, this simulation process is quite slow.

The Purify package[6] is a well-known proprietary package for detecting memory errors. Purify works by batch instrumentation of object files, reverse-engineering patterns in object code that represent compiled pointer operations, and replacing them with a mixture of inline code and calls into a runtime library.

2 How Mudflap Works

Mudflap works by inserting a pass into GCC's normal processing sequence. It comes after a language frontend, and before the optimizers, RTL expanders, and backend. It takes a restricted form of GCC *trees*, which are similar to abstract syntax parse trees, as input. It looks for tree nesting patterns that correspond to the potentially unsafe source-level pointer operations. These constructs are replaced with expressions that normally evaluate to the same value, but include parts that refer to libmudflap, the mudflap runtime. The compiler also adds instrumentation code associated with some variable declarations.

The purpose of this instrumentation is to assert a validity predicate at the use (dereferencing) of a pointer. The predicate is simply whether

[2] `http://gcc.gnu.org/projects/bp/main.html`

[3] `http://www-ala.doc.ic.ac.uk/~phjk/BoundsChecking.html`

[4] `http://immunix.org/stackguard.html`

[5] `http://developer.kde.org/~sewardj/`

[6] `http://www.rational.com/products/purify_unix/`

or not the memory region being referenced is recognized by the runtime as legal. If not, a violation is detected.

2.1 Object database

As a prerequisite for evaluating memory accesses, the runtime needs to maintain a database of valid memory objects. This database includes several bits of information about each object, which may be retained for some time even after it is deallocated.

- address range
- name, declaration source file, line number
- storage type (stack/heap/static/other)
- access statistics
- allocation timestamp and stack backtrace
- deallocation timestamp and stack backtrace

In order to update and search the object database, libmudflap exports a number of functions to be called by the inserted instrumentation. These basic ones include one to assert that a given access is valid, and a pair to add/remove a memory object to/from the database. These functions are passed pointer and size pairs plus some parameters to classify or decorate accesses and objects. For example, for a stack-based variable that may have pointer-based accesses would have a "register" call at the point of entry into its scope, and a corresponding "unregister" call when control leaves the scope. For heap-based variable, these calls would be performed within hooked allocation and deallocation primitives. For static variables, register calls are done early during program startup, and the effort of an unregister is not wasted during shutdown.

The database happens to be stored as a binary tree, naturally ordered based on the addresses of live objects. Its internal nodes are periodically rotated in order to move nodes for popular objects nearer to the root. There is a separate fixed-size array listing recently deallocated objects, used only during the performance-insensitive processing of violation messages.

During program startup, some selected objects are inserted into the database as special *no-access* regions. These represent address ranges that are certainly out-of-bounds for all instrumented programs, like the `NULL` area, and some libmudflap internal variables. Violations are always signaled when such objects are accessed by instrumented code.

2.2 Lookup cache

In order to hasten the database lookup, something which needs to be done many times, libmudflap maintains a *lookup cache*. This cache is compact global direct-mapped array, indexed using a simple hash function of the pointer value. Each entry in the cache specifies a memory address range that is currently valid to access. If an inline check of the cache for a given access is successful, the application avoids the call into libmudflap for the full-blown checking routine. Though the code may look complicated, it compiles down to a surprisingly small number of instructions.

```
/* uintptr_t: an integral type for
   pointers */
struct __mf_cache { uintptr_t low,
                    high; };
struct __mf_cache
       __mf_lookup_cache [];
uintptr_t __mf_lc_mask;
unsigned char __mf_lc_shift;

/* an approximation */
inline void
INLINE_CHECK (T* ptr, size_t sz, ...)
{
  uintptr_t low = ptr;
  uintptr_t high = low + sz - 1;
  unsigned idx =
    (low >> __mf_lc_shift) &
    __mf_lc_mask;
  struct __mf_cache *elem =
    & __mf_lookup_cache [idx];
  if (elem->low > low ||
      elem->high < high)
    __mf_check (ptr, sz, ...);
}
```

As with any caching scheme, choosing appropriate parameters (the *mask* and *shift* values) is a challenge. libmudflap has defaults suitable for mixed sizes of objects, which can be overridden by the user. In addition, when the runtime detects excessive cache misses, it adaptively tunes the cache parameters to better fit recent access patterns. For example, if accesses to small individual objects dominate, the current heuristic tends to decrease shift values. That way, more of the lower-order bits of the raw pointers remain to pick distinct cache lines. Section 3.1.4 lists libmudflap options that affect the lookup cache.

2.3 Instrumented expressions

Unsafe pointer expressions are easy to recognize when looking at C/C++ code. Systematically, most `*p`, `p->f`, and `a[n]` expression patterns need to be checked. Because mudflap operates in the middle of the compiler, we cannot look for such patterns in the source. Instead, we are given a representation of an entire function that resembles an abstract syntax tree. Expressions like the above are encoded in a web of nodes of GCC's `tree` type structure.

Mudflap traverses a function tree in program order, looking for certain pointer- or array-related constructs. These tree nodes are modified in place, replacing the simple pointer expressions with a GCC *statement expression*[7] that evaluates to the same value, but includes a call to the inline check routine outlined above. Shown in GCC's extended syntax, the expression `p->f` is changed roughly to `({check (p, ...); p;})->f`.

This in-place modification scheme supports recursion for nested constructs like `ptr->array[i]->field`. Here, two separate checks would be emitted: one for the element `ptr->array[i]`, and another to follow that pointer. The checks are performed in natural program order. Alternately, such nested constructs might be presented to mudflap already decomposed into an equivalent sequence of simpler expressions by the GIMPLE[8] transformations.

Table 1 shows the primitive expression patterns mudflap intercepts, and what address range is checked for each. For indirect accesses into larger compound objects, the checked range typically begins at the address of the outermost compound object, and ends by including the specific field or element being referenced. This way, the checked base value for similar accesses into the same structure or array can be constant, and take more benefit from the lookup cache. Notice that the checked range does *not* extend to include the entire compound object. This is because it is legal to allocate

[7]Statement expressions are a GCC extension that allows a brace-enclosed block to be treated as an expression. The last statement in the block is used as the expression value.

[8]`http://gcc.gnu.org/projects/tree-ssa/`

slightly less memory for a variable-sized structure than the raw sizeof, as long as the unallocated elements at the end are never accessed. GCC's own source code does this frequently.

2.4 Instrumented declarations

As discussed above, libmudflap's object database is kept up-to-date partly using instrumentation that tracks the lifetime of interesting memory objects. Some of these objects are variables declared as auto or static and have their addresses taken (or are indexed-into). For example, in the code segment below, the array variable needs to be registered with libmudflap (so the [i] indexing can be checked), but only for the duration of its scope (so that the returned pointer is invalid to dereference later).

```
char *foo (unsigned i) {
  char array [10];
  array [i] = 'a';
  return & array [i];
}
```

Tracking the lifetime of variables in a scope is tricky because control can leave a scope in several places. (Grossly, it might even enter in several places using goto.) The C++ constructor/destructor mechanism provides the right model for attaching code to object scope boundaries. Luckily, GCC provides the necessary facilities even to trees that come from the C frontend. There are several variants: the CLEANUP_EXPR node type, and the more modern TRY_FINALLY_EXPR. Both tree types take a block (a statement list) and another statement (a *cleanup*) as arguments. The former is interpreted as a sequence of statements such as any that follow a declaration within a given scope/block. The latter is a statement that should be evaluated whenever the scope is exited, whether that happens by break, return, or just plain falling off the end.[9]

We use this construct in mudflap by inserting one of these special try/finally tree patterns behind every declaration in need of lifetime instrumentation. The statement-list is the remainder of the original function, past the declaration in question, plus a register call for the declared object. The cleanup statement is an unregister call for the same object. The above function becomes the following, rendering TRY_FINALLY_EXPR in a Java-like way:

```
char *foo (unsigned i) {
  char array [10];
  try {
    __mf_register (array, 10, ...);
    array [i] = 'a';
    return & array [i];
  } finally {
    __mf_unregister (array, 10, ...);
  }
}
```

Mudflap also emits instrumentation to track the lifetime of some objects in the global scope: variables declared within file scope, or declared static within a function. This is done by intercepting assembler-related functions in gcc/varasm.c. It turns out at some literals like strings are like local static variables in this respect, so they too are registered. In each case, a list of declarations is accumulated until the end of the compilation unit. At that point a single dummy *constructor* function is synthesized, containing a long list of __mf_register calls. The linker arranges to call this and all other constructor functions early during the program startup.

2.5 Library interoperability

The above mechanisms are sufficient for checking pointer operations that are within an instrumented compilation unit. However, it is

[9]However, abrupt exit from a scope via a longjmp is not specifically handled at this time.

Sample declarations:

```
struct k {
  int a; /* offset 0 size 4 */
  char b; /* offset 4 size 1 */
}; /* size 8 */
int *iptr;
struct k *kptr;
char cbuf [];
short smtx [6][4];
int i, j;
```

expression	tree structure	check range	
		base	size
`*iptr`	`INDIRECT_REF(iptr)`	`iptr`	`4`
`*kptr`	`INDIRECT_REF(kptr)`	`kptr`	`8`
`kptr->a`	`COMPONENT_REF(INDIRECT_REF(kptr),a)`	`kptr`	`4`
`kptr->b`	`COMPONENT_REF(INDIRECT_REF(kptr),b)`	`kptr`	`5`
`cbuf[i]`	`ARRAY_REF(cbuf,i)`	`cbuf`	`i+1`
`smtx[i][j]`	`ARRAY_REF(ARRAY_REF(smtx,i),j)`	`smtx`	`8*i+2*j+2`

Table 1: Pointer expressions and their checked address ranges

often not possible to recompile an entire application, including the system libraries, with mudflap instrumentation. This means that several aspects of interoperability need to be addressed.

Most C/C++ programs make use of standard library functions (e.g., `strcpy`) that manipulate buffers given pointers. Typically, these libraries are not instrumented by mudflap, so they trust their arguments and don't perform pointer checking. An erroneous program can pass invalid pointers to these libraries, and bypass mudflap protection. libmudflap contains functions that interpose as a variety of such system library routines (though many more are yet to come). Each interposing function checks given buffer/length arguments, then jumps to the original system library. In this case, interposition is performed by replacing system library function names, via *preprocessor directives* implied by mudflap, with libmudflap names. This way, only instrumented object files are affected. Figure 2.5 shows a sample of this type of wrapper function in libmudflap.

In another scenario, an uninstrumented library may return to an instrumented caller some memory allocated from a shared heap. These memory regions should be registered with libmudflap, so that the instrumented code can be allowed to use them. Intercepting calls like `malloc` using preprocessor macros is not possible, since we are dealing with precompiled objects. We must intercept them at link time. Suitable mechanisms are available: *symbol wrapping* (for static linking with GNU ld) or *symbol interposition* (for shared libraries). libmudflap contains a protection mechanism to handle the case where a reentrant libmudflap⇒system-library⇒libmudflap call chain might occur.

```
void * WRAPPED_memmem (const void *haystack, size_t haystacklen,
                      const void *needle, size_t needlelen)
{
  INLINE_CHECK (haystack, haystacklen, __MF_CHECK_READ, "memmem haystack");
  INLINE_CHECK (needle, needlelen, __MF_CHECK_READ, "memmem needle");
  return memmem (haystack, haystacklen, needle, needlelen);
}

size_t WRAPPED_fread (void *ptr, size_t size, size_t nmemb, FILE *stream)
{
  INLINE_CHECK (ptr, size * nmemb, __MF_CHECK_WRITE, "fread buffer");
  INLINE_CHECK (stream, 1, __MF_CHECK_READ, "fread stream");
  return fread (ptr, size, nmemb, stream);
}
```

Figure 1: Sample libmudflap stdlib function wrappers

In yet another scenario, an uninstrumented library may return to an instrumented caller a value that points to some valid static data in the library. This could include objects as mundane as string literals. In this case, no link-time function interception can work, since these addresses are taken without reference to system functions. In order to tell automatically whether such a pointer is valid or not, libmudflap uses *heuristics*. These heuristics are checked when an access check is initially determined as a violation. They may look at other auxiliary platform-dependent data like the program's segment boundaries, stack pointer, and the like, to make a guess. Heuristics may be individually enabled or disabled at run time. See section 3.1.3 for more details.

2.6 Performance

Mudflap instrumentation and runtime costs extra time and memory. At build time, the compiler needs to process the instrumentation code. When running, it takes time to perform the checks, and memory to represent the object database. The behavior of the application has a strong impact on the run-time slowdown, affecting the lookup cache hit rate, the overall number of checks, and the number of objects tracked in the database, and their rates of change. Table 2 lists some of these. A few selected applications have been built with and without mudflap instrumentation, then run to estimate the slowdowns.[10] Table 2 also lists some applications, their performance factors, and associated slowdowns for a default mudflap build and run.

factor	description (+ polarity)
1	rare pointer manipulation
2	few large arrays
3	few addressed variables in scope
4	number cruncher
5	few tree/graph data structures
6	few objects in working set
7	non-changing access patterns

application	factors in effect		slowdown	
	+	−	build	run
BYTE nbench	3,4	1,2,5-7	3.5	3.5
spec2000 bzip2	2,5	1,3,4,6,7	4	5
spec2000 mcf	1-7		5	1.25

Table 2: Performance factors and overall measured slowdowns

[10] We used an x86 Linux host with ample memory, the same mudflap-capable compiler, and same optimization levels and linking modes.

2.7 Future

Mudflap development is ongoing; we anticipate several improvements. Significant performance benefits may arise from changing the instrumentation code (mainly for pointer checks), and functionality and performance benefits from the runtime.

We currently instrument each occurrence of a pointer dereference, even if that same pointer/size pair has been "recently" checked. Such checks could be eliminated if the compiler could prove that a subsequent check is redundant with respect to an earlier one. Extending from this, it may be possible to aggregate multiple checks based on the same pointer or array - imagine sequences of statements that access `ptr->field1` through `ptr->field5`. The compiler could create a single large check[11] near the beginning of a basic block, and eliminate subsequent checks for the same pointer/array. Some checks could be moved out of loops. In exchange for significantly better performance, such optimizations could detect pointer use errors out of program sequence.

Possible future libmudflap enhancements include support for multithreaded applications, growing the list of hooked functions to include more of the system libraries and system calls, more libmudflap entry points for use in an embedded system without a kernel, a better GDB interface, and general tuning.

3 Using Mudflap

Using mudflap is intended to be easy. One builds a mudflap-protected program by adding an extra compiler option (`-fmudflap`) to objects to be instrumented; one links with the same option, plus perhaps `-static`. One may run such a program by just starting it as usual.

[11]A large check would cover the maximal referenced range, including the last referenced field for a pointer, or the largest index for an array. This may require value range propagation or similar analysis.

In the default configuration, a mudflap-protected program will print detailed violation messages to `stderr`. They are tricky to decode at first. Figure 5 in the Appendix contains a sample message, and its explanation.

3.1 Runtime options

libmudflap observes an environment variable `MUDFLAP_OPTIONS` at program startup, and extracts a list of options. Include the string `-help` in that variable, and libmudflap will print out all the options and their default values. The display at the time of this writing is shown in Figure 5 in the Appendix. The next sections describe the options in groups.

3.1.1 Violation handling

The `-viol-` series of options control what libmudflap should do when it determines a violation has occurred. The `-mode-` series controls whether libmudflap should be active.

`-viol-nop` Do nothing. The program may continue with the erroneous access. This may corrupt its own state, or libmudflap's.

`-viol-abort` Call `abort()`, requesting a core dump and exit.

`-viol-segv` Generate a `SIGSEGV`, which a program may opt to catch.

`-viol-gdb` Create a GNU debugger session on this suspended program. The debugger process may examine program data, but it needs to quit in order for the program to resume.

`-mode-nop` Disable all main libmudflap functions. Since these calls are still tabulated if using `-collect-stats`, but the lookup cache is disabled, this mode is useful to count total number of checked pointer accesses.

`-mode-populate` Act like every libmudflap check succeeds. This mode merely populates the lookup cache but does not actually track any objects. Performance measured with this mode would be a rough upper bound of an instrumented program running an ideal libmudflap implementation.

`-mode-check` Normal checking mode.

`-mode-violate` Trigger a violation for every main libmudflap call. This is a dual of `-mode-populate`, and is perhaps useful as a debugging aid.

3.1.2 Extra checking and tracing

A variety of options add extra checking and tracing.

`-collect-stats` Print a collection of statistics at program shutdown. These statistics include the number of calls to the various main libmudflap functions, and an assessment of lookup cache utilization.

`-print-leaks` At program shutdown, print a list of memory objects on the heap that have not been deallocated.

`-check-initialization` Check that memory objects on the heap have been written to before they are read. Figure 5 explains a violation message due to this check.

`-sigusr1-report` Handle signal `SIGUSR1` by printing the same sort of libmudflap report that will be printed at shutdown. This is useful for monitoring the libmudflap interactions of a long-running program.

`-trace-calls` Print a line of text to `stderr` for each libmudflap function.

`-verbose-trace` Add even more tracing of internal libmudflap events.

`-verbose-violations` Print details of each violation, including nearby recently valid objects.

`-persistent-count=N` Keep the descriptions of N recently valid (but now deallocated) objects around, in case a later violation may occur near them. This is useful to help debug use of buffers after they are freed.

`-abbreviate` Abbreviate repeated detailed printing of the same tracked memory object.

`-backtrace=N` Save or print N levels of stack backtrace information for each allocation, deallocation, and violation.

`-wipe-stack` Clear each tracked stack object when it goes out of scope. This can be useful as a security or debugging measure.

`-wipe-heap` Do the same for heap objects being deallocated.

`-free-queue-length=N` Defer an intercepted `free` for N rounds, to make sure that immediately following `malloc` calls will return new memory. This is good for finding bugs in routines manipulating list- or tree-like structures.

`-crumple-zone=N` Create extra inaccessible regions of N bytes before and after each allocated heap region. This is good for finding buggy assumptions of contiguous memory allocation.

`-internal-checking` Periodically traverse libmudflap internal structures to assert the absence of corruption.

3.1.3 Heuristics

As discussed in Section 2.5, libmudflap contains several heuristics that it may use when it suspects a memory access violation. These heuristics are only useful when running a hybrid program that has some uninstrumented parts. Memory regions suspected valid by heuristics are given the special *guess* storage type in the object database, so they don't interfere with concrete object registrations in the same area.

`-heur-proc-map` On Linux systems, the special file `/proc/self/map` contains a tabular description of all the virtual memory areas mapped into the running process. This heuristic looks for a matching row that may contain the current access. If this heuristic is enabled, then (roughly speaking) libmudflap will permit all accesses that the raw operating system kernel would allow (i.e., not earn a SIGSEGV).

`-heur-start-end` Permit accesses to the statically linked text/data/bss areas of the program.

`-heur-stack-bound` Permit accesses within the current stack area. This is useful if uninstrumented functions pass local variable addresses to instrumented functions they call.

`-heur-argv-environ` This option adds the standard C startup areas that contain the `argv` and `environ` strings to the object database.

3.1.4 Tuning

There are some other parameters available to tune performance-sensitive behaviors of libmudflap. Picking better parameters than default is a trial-and-error process and should be undertaken only if `-collect-stats` suggests unreasonably many cache misses, or the application's working set changes much faster or slower than the defaults accommodate.

`-age-tree=N` For tracking a current *working set* of tracked memory objects in the binary tree, libmudflap associates a *liveness* value with each object. This value is increased whenever the object is used to satisfy a lookup cache miss. This value is decreased every N misses, in order to penalize objects only accessed long ago.

`-lc-mask=N` Set the lookup cache mask value to N. It is best if N is $2^M - 1$ for $0 < M \leq 10$.

`-lc-shift=N` Set the lookup cache shift value to N. N should be just a little smaller than the power-of-2 alignment of the memory objects in the working set.

`-lc-adapt=N` Adapt the mask and shift parameters automatically after N lookup cache misses. The adaptation algorithm uses the working set as identified by tree aging. Set this value to zero if hard-coding them with the above options.

3.2 Introspection

libmudflap provides some additional services to applications or developers trying to debug them. Functions listed in the `mf-runtime.h` header may be called from an application, or interactively from within a debugging session.

`__mf_watch` Given a pointer and a size, libmudflap will specially mark all objects overlapping this range. When accessed in the future, a special violation is signaled. This is similar to a GDB watchpoint.

`__mf_unwatch` Undo the above marking.

`__mf_report` Print a report just like the one possibly shown at program shutdown or upon receipt of `SIGUSR1`.

`__mf_set_options` Parse a given string as if it were supplied at startup in the `MUDFLAP_OPTIONS` environment variable, to update libmudflap runtime options.

4 Acknowledgments

The author thanks Ben Elliston for suggesting the mudflap name, Graydon Hoare for prototyping several parts of libmudflap, Diego Novillo for commiserating about GCC internals (and doing something to improve it), Red Hat (my employer) for funding mudflap's development, and future contributors for contributing in the future.

5 Availability

The source code of GCC with mudflap extensions, and of libmudflap, are available from the author, or by anonymous CVS. See `http://gcc.gnu.org/projects/tree-ssa/` for instructions.

```
mudflap violation 3 (check/read): time=1049824033.102085 ptr=080c0cc8 size=1
```

This is the third violation taken by this program. It was attempting to read a single-byte object with base pointer `0x080c0cc8`. The timestamp can be decoded as 102 ms after `Tue Apr  8 13:47:13 2003` via `ctime`.

```
pc=08063299 location=‘nbench1.c:3077 (SetCompBit)’
      nbench [0x8063299]
      nbench [0x8062c59]
      nbench(DoHuffman+0x4aa) [0x806124a]
```

The pointer access occurred at the given PC value in the instrumented program, which is associated with the file `nbench1.c` at line 3077, within function `SetCompBit`. (This does not require debugging data.) The following lines provide a few levels of stack backtrace information, including PC values in square brackets, and sometimes module/function names.

```
Nearby object 1: checked region begins 8B into and ends 8B into
```

There was an object near the accessed region, and in fact the access is entirely within the region, referring to its byte #8.

```
mudflap object 080958b0: name=‘malloc region’
bounds=[080c0cc0,080c2057] size=5016 area=heap check=1r/0w liveness=1
```

This object was created by the `malloc` wrapper on the heap, and has the given bounds, and size. The `check` part indicates that it has been read once (this current access), but never written. The liveness part relates to an assessment of how frequently this object has been accessed recently.

```
alloc time=1049824033.100726 pc=4004e482
      libmudflap.so.0(__real_malloc+0x142) [0x4004e482]
      nbench(AllocateMemory+0x33) [0x806a153]
      nbench(DoHuffman+0xd5) [0x8060e75]
```

The allocation moment of this object is described here, by time and stack backtrace. If this object was also deallocated, there would be a similar `dealloc` clause. Its absence means that this object is still alive, or generally legal to access.

```
Nearby object 2: checked region begins 8B into and ends 8B into
mudflap object 080c2080: name=‘malloc region’
bounds=[080c0cc0,080c2057] size=5016 area=heap check=306146r/1w liveness=4562
alloc time=1049824022.059740 pc=4004e482
      libmudflap.so.0(__real_malloc+0x142) [0x4004e482]
      nbench(AllocateMemory+0x33) [0x806a153]
      nbench(DoHuffman+0xd5) [0x8060e75]
```

Another nearby object was located by libmudflap. This one too was a `malloc` region, and happened to be placed at the exact same address. It was frequently accessed.

```
dealloc time=1049824027.761129 pc=4004e568
      libmudflap.so.0(__real_free+0x88) [0x4004e568]
      nbench(FreeMemory+0xdd) [0x806a41d]
      nbench(DoHuffman+0x654) [0x80613f4]
      nbench [0x8051496]
```

This object was deallocated at the given time, so this object may not be legally accessed any more.

```
number of nearby objects: 2
```

No more nearby objects have been found.
The conclusion? Some code on line 3077 of `nbench1.c` is reading a heap-allocated block that has not yet been initialized by being written into. This is a situation detected by the `-check-initialization` libmudflap option, referred to in section 3.1.2.

Figure 2: Sample libmudflap violation message, dissected

```
This is a GCC "mudflap" memory-checked binary.
Mudflap is Copyright (C) 2002-2003 Free Software Foundation, Inc.

The mudflap code can be controlled by an environment variable:

$ export MUDFLAP_OPTIONS='<options>'
$ <mudflapped_program>

where <options> is a space-separated list of
any of the following options.  Use '-no-OPTION' to disable options.

-mode-nop              mudflaps do nothing
-mode-populate         mudflaps populate object tree
-mode-check            mudflaps check for memory violations [default]
-mode-violate          mudflaps always cause violations (diagnostic)
-viol-nop              violations do not change program execution [default]
-viol-abort            violations cause a call to abort()
-viol-segv             violations are promoted to SIGSEGV signals
-viol-gdb              violations fork a gdb process attached to current program
-trace-calls           trace calls to mudflap runtime library
-verbose-trace         trace internal events within mudflap runtime library
-collect-stats         collect statistics on mudflap's operation
-sigusr1-report        print report upon SIGUSR1
-internal-checking     perform more expensive internal checking
-age-tree=N            age the object tree after N accesses for working set [13037]
-print-leaks           print any memory leaks at program shutdown
-check-initialization  detect uninitialized object reads
-verbose-violations    print verbose messages when memory violations occur [default]
-abbreviate            abbreviate repetitive listings [default]
-wipe-stack            wipe stack objects at unwind
-wipe-heap             wipe heap objects at free
-heur-proc-map         support /proc/self/map heuristics
-heur-stack-bound      enable a simple upper stack bound heuristic
-heur-start-end        support _start.._end heuristics
-heur-argv-environ     support argv/environ heuristics [default]
-free-queue-length=N   queue N deferred free() calls before performing them [4]
-persistent-count=N    keep a history of N unregistered regions [100]
-crumple-zone=N        surround allocations with crumple zones of N bytes [32]
-lc-mask=N             set lookup cache size mask to N (2**M - 1) [1023]
-lc-shift=N            set lookup cache pointer shift [2]
-lc-adapt=N            adapt mask/shift parameters after N cache misses [1000003]
-backtrace=N           keep an N-level stack trace of each call context [4]
```

Figure 3: List of libmudflap runtime options.