THIRD DEGREE

User Manual

24 May 1994

1. Introduction

If you don’t trust your code -- and you shouldn’t -- give it the ***Third Degree***.

Third Degree is a tool which performs memory access checks and memory leak detection of C and C++ programs at run-time. It accomplishes this by adding code to executables at link time, using ATOM. The added code performs run-time checks at all the interesting points: memory accesses, calls to memory allocators, procedure calls and returns, and program start and exit. The instrumented application is larger than the original application, runs exactly like it --just slower-- and in addition logs all errors and requested reports.

This instrumented program locates most occurrences of the worst types of bugs in C/C++ programs: array overflows, memory smashing and malloc/free errors. It also helps figure out the allocation habits of your application by listing the heap and finding memory leaks.

Third Degree attempts to make programs written in unsafe languages such as C and C++ somewhat safer. It cannot replace the guarantees that a truly safe language offers, such as bounds checking and safe pointer manipulation. ***Third Degree*** only catches errors which manifest themselves while the instrumented application runs, so it does not prove program correctness. It relies on a good test suite to exercise all bugs.

Third Degree instruments the whole program, including libraries, not just your code.

Third Degree is built on top of ATOM, and inherits any current limitations of ATOM. For example shared libraries cannot be instrumented; you must link your application so that it doesn’t use them.

Third Degree was designed for C and C++ programs, but will work on any program following the Alpha calling standard, including FORTRAN programs. ***Third Degree*** currently runs on Alpha under OSF. There are no plans to move it to other hardware platforms.

When you cannot use *Third Degree*

- If memory allocation is not based on *malloc* and its friends (*calloc, realloc*, *valloc*, *alloca* and *new* for C++ programs), ***Third Degree*** cannot follow heap allocations and deallocations, and will generate large numbers of false errors. In particular, calls of *sbrk* (other than sbrk(0)) and *brk* are detected and forbidden.
- If your program uses threads or coroutines, you’re out of luck. ***Third Degree*** simulates the program call stack. It will get very confused if the application uses more than one call stack. This might get fixed in future versions.
- Shared memory and mmap’ed memory is currently ignored in leak detection. If you keep the only pointer to a data structure somewhere else than in the stack or static area, ***Third Degree*** will consider it to be a storage leak.

2. Installing Third Degree

Third Degree is packaged with ATOM, version 0.5 or higher. If you have this manual, you probably already have ATOM on a machine near you, but you may want to get a newer version, or install it elsewhere. To check on releases of ATOM, look for release notes in the notes file WRL_ATOM on the machine CALDEC. Alternatively, you can look in the directory where ATOM and its documentation are kept:

```
> # see what versions are there:
> dls 'jove::/atom'
> # README describes what's there and how to install new ATOM versions
> dcp 'jove::/atom/README' .
> # user.ps is the ATOM user manual
> dcp 'jove::/atom/user.ps' .
> # ref.ps is the ATOM reference manual
> dcp 'jove::/atom/ref.ps' .
> # 3rd.ps is this manual
> dcp 'jove::/atom/3rd.ps' .
```

ATOM is distributed as an *setld* kit in *tar* form. The README file contains instructions for copying and installing it.

Using *Third Degree*

Third Degree comes with a driver program *3rd* to make it easy to modify your makefile. Just put the command "3rd" in front of your link command, and it will invoke ***Third Degree*** to instrument the executable:

```
3rd cc test.c -o test
```

In most makefiles, just redefine the C-compiler macro:

```
CC = 3rd cc
```

GNU software can be built with ***Third Degree*** by setting the CC environment variable before running configure:

```
setenv CC "3rd cc"
./configure
```

The *3rd* program takes the following flags, which must precede the compilation command:

- -v: print all commands as they are executed
- -V: print all commands, but do not execute them
- -tool t: instrument the program with ATOM tool *t* instead of ***Third Degree***.

Any other flag is passed on verbatim to ATOM. For example:

```
3rd -A1 -tool gprof cc test.c -o test
```

will use the 3rd driver to instrument test with the profiler gprof, using ATOM'S low-overhead instrumentation technique.

Using ATOM Explicitly

Third Degree can also be run like any other ATOM tool.

Compile	`cc -c $*.c -o $*.o`
Pre-link	`cc -Wl,-r -non_shared x.o y.o z.o ... -o app.rr` This pre-links your application, retaining relocation records and not using shared libraries. Notice that `-Wl,-r` is a single option, with no space.
Instrument	`atom -A1 app.rr -tool 3rd -o app.3rd` ATOM adds the instrumentation specified by ***Third Degree***. The flag -A1 selects a low-overhead instrumentation technique.
Run	The executable *app.3rd* should behave exactly like *app*, except that it is a larger executable, runs slower, and uses more memory. Currently the instrumented executable cannot be used with a symbolic debugger, even if you compile with -g. Oh yes, it also writes a useful log *app.3log*.
Browse the log	To make sense of it, or to specify various options in a *.3rd* file, you may have to read the rest of this manual. You may skip to the last section, listing of all options for *.3rd*, in order to whet your appetite.

3. Step-by-step Example

We guide you through a small example in order to give you a taste of ***Third Degree***. Let's assume that we want to debug a small application, *ex.c* shown below:

```
/* ex.c */
#include <assert.h>

int Bug() {
    int q;
    return q;           /* q is uninitialized */
}

long* Booboo(int n) {
    long* t = (long*) malloc(n * sizeof(long));
    t[0] = Bug();
    t[0] = t[1];        /* t[1] is uninitialized */
    t[n] = n;           /* array bounds error*/
    if (n<10) free(t);  /* may be a leak */
    return t;
}

main() {
    long* t = Booboo(20);
    t = Booboo(4);
    free(t);            /* already freed */
    exit(0);
}
```

Modifying the Makefile

After installing ***Third Degree***, we modify the Makefile to look like this:

```
CC = 3rd cc

.c.o:
        ${CC} -c -I. $*.c -o $*.o

ex.3rd: ex.o
        ${CC} ex.o -o ex.3rd
```

Now we can make *ex.3rd*.

```
> make ex.3rd
3rd cc -c -I. ex.c -o ex.o
3rd cc ex.o -o ex.3rd
> ex.3rd
>
```

Customizing Third Degree

A customization file named *.3rd* is used to turn on and off various capabilities of the tool, and lets advanced users set internal parameters. A list of all options is given in section 6. The file we use here is very short and almost self-explanatory. It specifies that we want to be informed of memory access errors, and of all memory leaks found at the end.

```
memory_errors   yes
all             leaks   at_exit
heap_history    yes
```

Now run the instrumented application *ex.3rd* and check the log *ex.3log*. This file contains several sections, which we now take in sequence.

Copy of the .3rd file

Third Degree looks for a *.3rd* file first in the local directory, then in your home directory. If no *.3rd* file is found, you'll be reminded. It's fine to omit this file, and you'll simply get the default settings: list memory errors, detect leaks at program exit. If you have a *.3rd* file, ***Third Degree*** copies it to the log file:

```
//////////////  begin .3rd  ///////////////////
-----------------------------------------------
memory_errors   yes
all leaks       at_exit
heap_history    yes
-----------------------------------------------
//////////////  end  .3rd ///////////////////
```

List of run-time memory access errors

Various types of errors can be found at run-time: reading uninitialized memory, reading/writing unallocated memory, freeing invalid memory, and a few really nasty error conditions likely to cause an exception. Each error is numbered and carries the process id, in case the program forked several processes.

```
---------------------------------------------------- pid=3878 -- rus -- 0 --
Reading uninitialized memory in stack frame of Bug at offset 8
  - Bug                                        ex.c, line 6
  - Booboo                                     ex.c, line 11
  - main                                       ex.c, line 19
```

The first error is quite simple: a local variable of procedure *Bug* is being read before being initialized. The line number confirms that indeed, *q* is never given a value.

```
---------------------------------------------------- pid=3878 -- ruh -- 1 --
Reading uninitialized memory in heap at 0x140031a18
  at byte 8 of 160-byte block at 0x140031a10
  - Booboo                                     ex.c, line 12
  - main                                       ex.c, line 19

This block was allocated by malloc at
  - Booboo                                     ex.c, line 10
  - main                                       ex.c, line 19
```

The error happened on line 12: *t[0] = t[1]*. Since the array is not initialized, the program is indeed reading uninitialized memory in the heap. The object is characterized by the call stack which allocated it. We always print the entire call stack, no matter how deep it is.

```
---------------------------------------------------- pid=3878 -- wih -- 2 --
Writing unallocated memory in heap at 0x140031ab0
  1 byte beyond 160-byte block at 0x140031a10
  - Booboo                                     ex.c, line 13
  - main                                       ex.c, line 19

This block was allocated by malloc at
  - Booboo                                     ex.c, line 10
  - main                                       ex.c, line 19
```

This error is more serious than the previous one: the program has written one position past the end of the same array, potentially smashing innocent values or worse, ***Third Degree*** internal data structures. Keep in mind that later errors could be a consequence of this error.

```
---------------------------------------------------- pid=3878 -- fof -- 3 --
Freeing already freed memory in heap at 0x140038010
  at byte 0 of 32-byte block at 0x140038010
  - main                                       ex.c, line 21

This block was allocated by malloc at
  - Booboo                                     ex.c, line 10
  - main                                       ex.c, line 20

This block was freed at
  - Booboo                                     ex.c, line 14
  - main                                       ex.c, line 20
```

For errors involving *free*, usually three call stacks are given: where the error occurred, where the object was allocated, and where it was freed. Upon examining the program, one will see that the second call of Booboo (line 20) indeed frees the object (line 14), which we then attempt to free again (line 21).

Memory Leaks

Since in this example we have requested a list of all memory leaks upon program exit, we get a list, organized by call stack.

```
-----------------------------------------------------------------------
  Searching for all leaks in heap before 1st call of _exit in pid=3878
  We found 160 bytes in 1 object:

160 bytes in 1 leak (including 1 super leak) created by malloc at
  - Booboo                                      ex.c, line 10
  - main                                        ex.c, line 19
```

All true! The first call of Booboo did not free the object, and nobody else did, so this is a leak.

In addition, there is no pointer anywhere to this object, so it qualifies as a 'super leak'. The distinction is often useful to find the real culprit for large memory leaks. Consider a large tree structure and assume the pointer to the root has been erased. Every object in the structure is a leak, but losing the pointer to the root is the real cause of the leak. Since all objects but the root still have pointers to them - albeit only from other leaks - only the root will be identified as a super leak, and therefore the likely cause of the memory loss.

Heap History

This is an experimental feature that could change a lot in future versions depending on the feedback we get from you. The idea is to provide a summary of the contents of all objects allocated at different points in the programs. For each call point to a memory allocator in the program, the contents of the objects it allocates are examined when they are freed or at program end to show how effectively the memory was used. This feature can be used to eliminate unused fields in data structures, and to pack them for more efficient memory usage. It is enabled by the *heap_history* command.

```
_______________________________________________________________________
-----------------------------------------------------------------------
                Heap Allocation History for pid=3878
_______________________________________________________________________
-----------------------------------------------------------------------
                                  type size; one char per 32-bit
                                   '.' never written
                                   'i' written
                                   'z' zero
       total    total  % bytes     'pp' pointer
       bytes  objects  written     'ss' sometimes pointer

         192        2      8.3  Booboo
                                ex.c, line 10
                                32 bytes (variable size)
                                zz......
```

The sample listing below shows one entry per allocation point, i.e. a file and line number where an allocator (*malloc* or friends, or *new*) was called. Each entry shows the total number of bytes allocated at this point; how many objects were allocated; the fraction of the bytes which were actually written; the procedure, file name, and line number; and some information about the type allocated there. Entries are sorted by decreasing volume of allocation.

Our sample program has indeed allocated 2 objects, for a total of 8*(20+4) = 192 bytes. Only the first word (8 bytes) of each array was written, leading to a ratio of 2*8/192 = 8.33%. The first quad word was found to contain 0 in all cases, while the rest of the object was not written.

Notice that even if an entry is listed as allocating 100M bytes, it does not mean that at any point in time 100M bytes of heap storage were used by objects allocated there: it is a cumulative figure. Over the lifetime of the program, this point has allocated 100M bytes, which may have been freed, may have leaked, or might still be in the heap. The figure simply indicates that this allocator has been pretty active.

The fraction of the bytes actually written should ideally always be close to 100%. If it's much lower, consider this as a warning: some of what is allocated is never used! We have encountered several situations were the percentage is low:

- A large buffer was allocated, but only a small fraction was ever used.
- Parts of every object of a given type are never used. They may be forgotten fields, which should be taken out, or padding between real fields. This problem is quite serious on the Alpha, where the 64-bit alignment rules can cause a fair amount of memory to be wasted in structures. This kind of problem can arise often in porting from a 32-bit architecture.
- Some objects have been allocated, but never used. They may not show up as leaks if they are kept on a list. Maintaining free lists of objects can cause this.

The last part of each entry is an attempt to reverse-engineer the type of what was allocated at this point. To do this, we watch objects as they are freed, and we also scan for active objects or leaks left in the heap at exit. We first print the size of objects allocated from this point: if the objects allocated do not all have the same size, we consider the minimum size, and signal the fact with *(variable size)*. We then print a summary of the findings about the type, based on tags and values. There is one character per 32-bit longword, lsb to msb.

- A dot '.' shows a longword which is never written, in any of the objects. It is a definite sign of wasted memory. Further analysis is generally required to see if it is simply a deficiency of the test which truly never used this field; if it is a padding problem solved by swapping fields or choosing better types; or if this field is obsolete.
- An 'i' corresponds to a longword which was written, at least in one object of this type.
- A 'z' is a field whose value was always 0 when the object was sampled. Similarly, 's', for sign extension, is used for longwords always sampled as '0' or '-1'.
- A 'pp' indicates a pointer, i.e. a 64-bit quantity which looked like a valid pointer, or 0, on every object.
- A 'ss', or sometime pointer, looked like a pointer at least in one of the objects, but not in all objects. It could be a pointer which is not initialized in some instances, or a field which by pure luck matches a pointer in rare circumstances, or a union...

Statistics

This section summarizes the memory used by the program by size and address range.

```
----------------------------------------------------------------------
                General Statistics for pid=3878
----------------------------------------------------------------------
          stack       1616 bytes [0x11ffff9b0..0x120000000]
           text      24576 bytes [0x120000000..0x120006000]
      init data       8192 bytes [0x140000000..0x140002000]
    uninit data      15712 bytes [0x140002000..0x140005d60]
           heap     245860 bytes [0x140005d60..0x140041dc4]
```

4. Instrumenting the Heap

In addition to run-time checks which make sure that only properly allocated memory is accessed and freed, ***Third Degree*** provides two ways to understand what is going on in the heap: it can find memory leaks and list the contents of the heap. By default, if you don't provide a ***.3rd*** file, ***Third Degree*** will check for leaks when the program exits.

Memory Leaks

A memory leak is an object in the heap to which there is no pointer. It can thus no longer be accessed, and can no longer be used or freed. It is useless, and will simply never go away!

We find memory leaks using a simple *trace-and-sweep* algorithm: starting from a set of roots (currently the registers, the stack and the static area), find pointers to objects in the heap and mark these objects as visited. Then recur on all potential pointers inside these objects. Finally, sweep the heap and report all unmarked objects: these are the leaks.

The trace-and-sweep algorithm finds all leaks, including circular structures. This algorithm is conservative: in the absence of type information, any 64-bit pattern properly aligned and pointing to the beginning of a valid object in the heap is treated as a pointer. This can lead to various problems:

- Any bit pattern (string, integer, floating-point number, packed struct) looking like a heap pointer can be confused with a true pointer, and the tool can miss a true leak. As experience with conservative garbage collectors shows, this is very rare, and 64-bit pointers help make this even rarer.
- Hiding true pointers by storing them in the address space of some other process or by encoding them is a dirty trick. It confuses the leak detector which then reports spurious leaks. To help with this, the option ***pointer_mask*** lets you specify a mask which is AND-ed with every potential pointer; for example, if you use the top 3 bits of pointers as flags, specify a mask of 0x1fffffffffffffff.
- We consider pointers either to the beginning of an object, or to its interior, as valid, so only objects with no pointers to any address they contain are considered leaks.

Heap and Leaks Reports

Both reports can be done incrementally, listing only what is new since the last report of this type, or give a complete list. They can be requested when the program terminates, or before/after every N^{th} call of a specified function.

In the report, memory leaks (objects) are listed by decreasing importance in number of bytes. Objects allocated with identical call stacks are grouped together. So if the same call sequence in the program allocates a million one-byte objects, they will be reported as a one-megabyte group containing a million allocations.

You may wish to tell ***Third Degree*** when objects are the same and should be grouped in the report, or are different and should not. To do this, you can set the depth of the call stack used to differentiate leaks or objects. For example, if a depth of 1 is used for objects, valid objects in the heap will be grouped by the function and line number which allocated them, no matter what function was the caller. Conversely, a very large depth for leaks will ensure that only leaks allocated at points with identical call stacks from *main* upwards will be reported together.

In most heap reports, the first few entries account for most of the storage, but there is a very long tail of small entries. To limit the length of the report, you can define a threshold as a percentage of the total memory leaked (in use). When all smaller remaining leaks (objects) amount to less than this fraction of the total, they are grouped under a final entry.

Fine points:

- Because *realloc* always allocates a new object (malloc/copy/free), it can lead to counter-intuitive behavior. An object can be allocated, listed, shrunk through a call of realloc, and be listed again under a new identity.
- Leaks and objects are mutually exclusive: an object must be reachable from the roots.

Hints for Leak Hunting

It may not always be obvious when to search for memory leaks. ***Third Degree*** checks for leaks at program exit by default, but this may not always be what you want.

Leak detection is best done as near as possible to the end of the program while all used data structures are still in scope. But remember that the roots for leak detection are the contents of the registers, stack and static areas. If your program terminates by returning from *main*, and if the only pointer to one of its data structures was kept on the stack, this pointer will not be seen as a root during the leak search, leading to false reporting of leaked memory. In this example,

```
main (int argc, char* argv[]) {
    char* bytes = (char*) malloc(100);
    exit(0);
}
```

putting

```
all leaks at_exit
```

in the *.3rd* file will not find any leaks, because the program exits with *main's* variables still in scope. In this example, however,

```
main (int argc, char* argv[]) {
    char* bytes = (char*) malloc(100);
}
```

the same leak check may report a storage leak, because *main* has returned by the time the check happens. Either of these two behaviors may be correct, depending on whether *bytes* was a true leak, or simply a data structure still in use when main returned.

Rather than reading the program carefully to understand when leak detection should be done, you can check for new leaks every so many memory allocations, how often of course depending on the particular program. The following commands could be put in the *.3rd* file for C and C++ programs:

```
# use this for C programs
no leaks at_exit
new leaks before malloc every 10000

# use this for C++ programs
no leaks at_exit
new leaks before "operator new(unsigned long)" every 10000
```

This strategy will not check for leaks in the last 9999 allocations before the program exits, and may slow down execution substantially if the frequency of checking for leaks is too high. Note also that *malloc* is not the only way C and C++ programs allocate memory: there is also *realloc*, *calloc*, *valloc* and *alloca*, so the correct allocator to instrument may vary between applications.

In short, finding the correct place to search for leaks may require some thought. It is best done as near as possible to the end of the program while all used data structures are still in scope, but if there is no good procedure entry or exit to instrument, periodic leak checking may be an acceptable alternative.

5. How Third Degree Works

What is checked

Conceptually, every memory access is preceded by a procedure call which checks whether the access is valid. In reality, analysis of each procedure allows us to target only a fraction of the loads/stores for instrumentation, without losing any safety.

Third Degree implements a tagged memory, i.e. it associates a state with every (32-bit) longword indicating whether this memory location is valid or not. Any attempt to read or write invalid memory is reported as an error. Only memory in the stack and heap is tagged; static variables are always 'valid', so we do not tag them. By default memory is invalid. The content of a block returned by *malloc*, *realloc*, *valloc*, *calloc*, *alloca* and *new* is **valid**. Once freed, it goes back to being **invalid**. In addition, every such block is surrounded by some invalid memory, so any access a few words before or after the object can be caught. Similarly, the portion of the stack used by a procedure frame is marked **valid** upon entry, and **invalid** upon exit. Certain words in a frame, such as the register save area and return address, nevertheless remain **invalid** to help detect array bounds errors in the stack.

When a stack or heap location first becomes valid, its contents is set to a particular *uninitialized* value: 0xfff8a5a5fff8a5a5 if you must know. This value is the same for all newly allocated locations, and it is chosen to be an unlikely integer, an invalid pointer, an invalid floating point number, and a non-printing character string, both on 64 and 32 bits. If a load reads this value, an error is reported.

Memory Allocation

In order for ***Third Degree*** to work, your application must allocate memory using *malloc*, *realloc*, *valloc*, *calloc*, *alloca* and *new* exclusively. Any custom scheme which calls *sbrk* and *brk* directly cannot be instrumented properly, and causes the tool to terminate with an exception.

Be also aware that each object thus allocated is actually padded with information on both sides, and thus uses more memory than in your original application: there is a fixed header (16 bytes) and a user-settable tail padding (16 bytes by default). In the case of *valloc*, a whole page is used for the header in order to satisfy the alignment constraint.

Free Queue

In order to catch accesses to freed objects, or multiple calls of free() on an object, ***Third Degree*** makes sure that data is not re-used immediately after being released by a call to *free*. It is thus first stored in a queue; only after the queue size exceeds a user-settable value do the oldest members get freed for good.

Granularity

Since we associate a tag with each long-word (32 bits), some program errors may not be detected. For example writing in the byte following a 5-byte object is not detected, but writing in the 9th byte is detected. Clearly, associating a tag with each byte would catch more errors, though it would use a lot more extra memory to do so, but on the Alpha we don't have a choice: the reason is the absence of instructions accessing bytes or words (16 bits) directly. The Alpha instructions for memory access fall in three categories: longword (32 bits) instructions, used to manipulate ints and floats; quadword (64 bits) instructions, used to manipulate longs and doubles; unaligned quadword instructions, which accept arbitrary byte addresses.

Any byte manipulation is done through masking and shifting of quadwords. It could be hoped that any sequence of instructions intending to read or modify a quantity smaller than 32 bits would use an unaligned instruction, thereby giving a hint about which byte(s) is the real target. Alas, enough examples were found where regular quad instructions were used to actually access a single byte that we had to abandon this idea.

6. The .3rd file

Third Degree looks for the option file *.3rd* in the local directory, then in your home directory.

If no file is found, the defaults listed here apply. Notice that for boolean options, yes can be omitted, i.e. 'heap_history' is the same as 'heap_history yes'.

memory_errors	**yes**/**no**	yes
By default, memory access errors are listed; fatal errors always reported.		
all/new objects/leaks	**at_exit**	
To obtain the report when the program exits.		
all/new objects/leaks	**before**/**after** proc_name **every** N	
To obtain a report before or after every N^{th} call of this procedure. C++ procedure names may be enclosed in single or double quotes.		
no leaks	**at_exit**	
To disable the default leak check at program exit.		
object_stack_depth	integer	10000
leak_stack_depth	integer	10000
Controls the depth of the call stack used to distinguish objects.		
object_min_percent	float	1.0
leak_min_percent	float	1.0
Entries amounting to less that this percentage of a report are grouped. The number is a floating-point number interpreted as a percentage.		
mail_to	myself@you_know_where.com	
Send a copy of the log file to this address. Use several command lines for multiple addresses.		
ignore	error	
ignore	error proc_or_file_name	
ignore	error proc_or_file_name **line** integer	
Uses a three-letter error name (see below) and a procedure or file name. C++ procedure names may be enclosed in single or double quotes. File names should not include directory names. If no file or proc name is present, this error is disabled everywhere.		
pointer_mask	64-bit hex number	0xffffffffffffffff
Used as a mask on potential pointers when searching leaks.		
object_padding	integer	16
Each heap-allocated object has a padding of this many bytes after it. A larger value may help catch more errors but uses more memory.		
free_queue_bytes	integer	1000000
The purgatory where freed blocks spend some time before being truly freed is a queue of at most ***free_queue_bytes***. Increasing the size of the queue uses more memory, but helps to find problems of access to freed objects.		
heap_history	**yes**/**no**	no
include	file_name	
Read more commands from this file.		

comment lines start with a '#'

Errors

Errors fall in two categories: memory access errors, and fatal errors. Fatal errors, which cannot be turned off, include:

- pop not matching push: probably a data-structure corruption. Check to see if an invalid write was reported earlier in the log.
- long jump to nowhere: ditto.
- bad parameter: for example *malloc(-10)*.
- failed allocator: malloc or a similar function has returned 0; no more memory!
- call of sbrk: sorry, your program cannot call sbrk directly with a non-zero argument.

Fatal errors cause the instrumented application to crash, **after** flushing the log file. If the application crashes, check the log file first. Remember, symbolic debuggers cannot currently be used on instrumented programs.

Report of memory errors can be suppressed by specifying an ***ignore*** command in the *.3rd* file. This silences the reporting of these errors in every call of the specified function. The syntax uses a three-letter name for the error. A list of errors with their short name follows:

ror	reading out of range, neither in heap, stack or static area.
ris	reading invalid data in stack: probably an array bound error.
rus	reading an uninitialized (but valid) location in stack.
rih	reading invalid data in heap: probably an array bound error.
ruh	reading an uninitialized (but valid) location in heap.
wor	writing out of range, neither in heap, stack or static area.
wis	writing invalid data in stack: probably an array bound error.
wih	writing invalid data in heap: probably an array bound error.
for	freeing out of range, neither in heap or stack.
fis	freeing an address in the stack.
fih	freeing an invalid address in the heap: no valid object there.
fof	freeing an already freed object.
fon	freeing a null pointer (really just a warning).
mrn	malloc returned null.

Fix and Retry

If your instrumented program suffers from write errors, specially heap write errors, we suggest that you fix the first few ones and try again rather than trying to make sense of the whole error log. Not only can errors compound, but the internal data structures of ***Third Degree*** could be corrupted by these invalid writes: there is really no safe place to hide from a wild write!

But it used to work...

Our technique for detecting the use of uninitialized values can cause programs which work to fail when instrumented. For example if a program depends on the fact that the first call of malloc returns a block initialized to zero, the instrumented program will fail since all blocks are initialized by ***Third Degree*** to a rare, definitely non-zero, value. This style of programming is unclean, and should be fixed, in that case by using calloc instead of malloc.

Ignoring errors

If you get annoyed by library functions for which you do not have the source, it might be a good idea to use the ***ignore*** option to silence certain errors. Most will be uninitialized stack reads. See the ***False Positives*** section below for an explanation of why they arise. Really serious errors, like writing past the end of an array in the heap or stack, should not be silenced: instead, report them to the person writing the library and get them fixed!

False Positives

Even though we made a serious effort to curb their appearance, spurious errors can still clutter the log. There are several sources:

- Compilers sometimes generate loads of uninitialized local variables when you might not expect them. Errors in library functions can be ignored, but errors in your code could be real, as in the example of chapter 2. Compiling with a non-default optimization setting seems to increase the chances of these code sequences.
- Copying a partially uninitialized structure is currently reported as an error. It is a good subject of debate whether copying junk and then ignoring it is a bug or not. For the moment, we are conservative. Silencing errors about uninitialized reads in strcpy, bcopy and memcopy should help. We would not dare silence errors about write, by the way.
- Hiding pointers, encoding them, keeping only pointers to the inside of a heap object will fool the leak finder. The same rules as for conservative garbage collectors apply here.
- There is still the infinitesimal chance that a program generates the magic value used to encode 'uninitialized', and ***Third Degree*** reports a false error. Frankly, the chances are very low, since this value is not a pointer, a floating-point number, a remarkable integer or a printable string, on 32 or 64 bits. Don't lose too much sleep over this one...

The first category of error is the most frequent, and it is always caused by uninitialized local variables. The procedure *_doprnt*, called from *printf* and *sprintf*, for instance, has one of these, and you will probably encounter it sooner or later. Sometimes the error may not be obvious at first. Consider the following procedure and error:

```
 1  int CMove(int arg1, int arg2) {
 2      int lt, gt, neg;
 3      if (arg1<0 && arg2<0) {
 4          lt = arg1<arg2;
 5          neg = 1;
 6      } else {
 7          gt = arg1>arg2;
 8          neg = 0;
 9      }
10      /* at this point only one of lt and gt has a value */
11      return (neg) ? lt : gt;
12  }
```

```
------------------------------------------------------- pid=4102 ------- 0 --
Reading uninitialized memory in stack frame of CMove at offset 0
  - CMove                                       iwt.c, line 11
  - main                                        iwt.c, line 345
```

The reason for the error is that at line 11, only one of *lt* and *gt* has been assigned a value, but the code sequence for the return statement loads both values and selects one with a conditional move instruction. The cure in such cases is to ensure that all local variables have values at the point in the code where the error is reported.

False Negatives

Third Degree can miss real errors.

- Accessing the wrong object in the heap by chance cannot be detected: we only find memory accesses out of objects. For example if *a[last+100]* is the same address as *b[0]*, ***Third Degree*** will miss this. One can change the chances of this happening by altering the amount of padding added to objects. To do this, change the value of ***object_padding*** in the *.3rd* file.
- Walking past the end of an array by fewer than 8 bytes may be missed, because of the granularity of memory tags. In the heap, objects are surrounded by "guard words" and only small array bounds errors will be missed. In the stack, adjacent memory is likely to contain local variables, and larger bounds errors may go undetected. For example sprintf'ing in a way-too-small local buffer will probably be caught, but if the array bounds are only exceeded by a few words, and enough local variables surround the array, the error can go undetected.
- As explained above, we can miss a memory leak if some random bit pattern looks like a valid pointer to this leaked object.

Acknowledgments

Third Degree was written by Jeremy Dion and Louis Monier of Digital's Western Research Lab. It was inspired by OM, the link-time code modification system developed at WRL by Amitabh Srivastava, and ATOM, its programmable instrumentation interface written by Alan Eustace, also of WRL. The symbolic interpretation techniques used to reduce the number of instrumentation points is due to Francois Bourdoncle of Digital's Paris Research Lab. Andy Payne of Digital's Cambridge Research Lab, in addition to giving much constructive feedback, has contributed the driver *3rd* for ***Third Degree***. Evelyne Monjoin came up with the name ***Third Degree***.