Identifying Dynamic Memory Errors in C Programs on AIX – Improving Software Reliability

Scott Trent
trent@jp.ibm.com

```
C Code.
C Code Run.
Run Code Run...   Please!
```

- anonymous

Summary

All experienced software developers have war stories to tell about challenging bugs they have encountered. A reoccurring and frustrating theme among these software veterans is bugs that seem to materialize and vanish without apparent cause. These are nearly always caused by dynamic memory errors. Complete with original sample code, this paper describes these errors, and provides solutions and concrete examples that can be used by developers to fix these bugs with minimal effort. Applied throughout the development process, these techniques will improve software quality, reliability, customer satisfaction, and result in multiple cost savings.

要約

経験豊かなソフトウエア開発家は皆、解決困難なバッグに苦労させられたことがあるだろう。それらのバッグが理由なく現れたり、また消えたりするというのはよくある現象である。この原因は殆ど動的メモリ扱いのエラーである。オリジナル・サンプル・コードを含め、これらのバッグを理解し、楽に解決するためのソリューションや具体的な例を検討する。この論文に紹介するテクニックを開発プロセスに取り入れた場合、ソフトウエアの品質や信頼性の改善、お客様の満足度の向上、経費節減、等が期待できる。

1.0 Introduction

1.1 The Stakes

Any experienced worker in the computer industry is deeply aware of the vital importance of continually learning new technologies and methodologies. The techniques and knowledge described in this paper will significantly improve a programmer's ability and productivity when faced with a program suffering from a dynamic memory bug.

As highlighted in the table below, in today's network connected, e-commerce business environment, the price of downtime resulting from unreliable software can not be ignored. Fortunately, integrating the techniques discussed in this paper into a software development process can also easily reduce the number of programming errors, as heretofore nearly impossible to detect dormant programming errors become clearly visible. This directly results in improved software quality, reliability, and thus higher customer satisfaction as software related downtime is reduced.

The Cost of Unplanned System Outage

Industrial Application	Cost per Hour of Downtime
Retail Brokerage	$6.50 Million USD
Credit Card Sales Authorization	$2.6 Million USD
Pay-per-view Television	$1.1 Million USD
Package Shipping Services	$200,000 USD
Airline Reservations	$115,000 USD
Catalog Sales	$90,000 USD
Infomercial/toll-free Telephone Promotions	$90,000 USD

Sources: IBM Redbook SG24-2018, Gartner Group, Contingency Planning Research, Inc

In the past, dormant memory errors were often found by the Customer in unusual circumstances. In addition to improving software quality and reliability, and improving customer satisfaction, the techniques presented in this paper can be used to locate and fix software dynamic memory related defects earlier in the development phase. This will result in a direct cost savings, as demonstrated by the following chart. Specifically, the use of these techniques has identified at least 163 defects in development phase of AIX Operating System Development, yielding a theoretical savings of $2.5 Million USD.

Defect Cost over Time

Development Phase	Cost per Defect
Coding	$25 USD
Unit Test	$130 USD
Function Test	$250 USD
Field Test	$1000 USD
Post Release	$15,000 - $40,000 USD

Source: Applied Software Measurement, Capers Jones, 1996

1.2 The Challenge

Although easy to understand and remove when finally identified, dynamic memory errors in C are one of the most challenging kinds of programming problems to cope with. Unlike classes of programming errors that are easily detected with unit testing or code inspection, many subtle

dynamic memory bugs lay dormant long after they are originally introduced, since they erroneously modify memory that is not commonly used. When this memory is eventually used due to a future program modification, the use of a rare code path, or even a simple change in environment, the previously dormant error can cause a cascade effect so that the end-user experiences a problem that is quite removed from the actual root cause.

Since this cascade effect is dependent upon the exact layout of memory and data within a process, the common actions that a programmer takes to locate a bug such as adding `printf()` statements, building the program in debug mode, running in a different environment, etc., can make the problem disappear even though the error has not actually been fixed. These kinds of programming errors are very difficult and frustrating for a programmer to locate with normal debugging methodology.

This paper will introduce the major categories of dynamic memory handling errors that programmers tend to make, along with a description of the specialized tools and techniques that are effective in resolving these dynamic memory problems. In fact, not only has the author personally used each of showcased tools in AIX Operating System Development at IBM-Austin to improve AIX code quality, but he has also observed their overall use and benefit in AIX Development. These tools have been useful in resolving over 300 defects in AIX Development. Specific usage examples and output, screen shots, original and complete code samples, references, etc. are included to make this paper a valuable tool for C programmers of all experience levels. The use of these tools and techniques in both testing and debugging phases of software development will increase productivity as well as improve software quality and reliability.

2.0 Dynamic Memory Programming Errors

Memory used to store variable and constant data within a process on AIX loosely falls into three categories.

(1) Global variables and constants initialized by the compiler. (Also known as BSS and Data segments.) This memory region is statically sized.
(2) Variables defined within functions are stored on and removed from the stack depending upon the current scope. The stack is a variable sized region, which in a default 32-bit process starts at 0x2FFFFFFC and grows down..
(3) Dynamically requested memory from `libc.a` functions such as `malloc()`, `realloc()`, `alloca()`, or `valloc()` is known as the “heap”. The heap is a dynamically sized region.

. Of these three types of memory, management of data in the heap is recognized to be the source of most dynamic memory handling programming errors. Thus modern programming languages such as Java have been designed so that dynamic memory (i.e. the heap) is not explicitly maintained by the programmer. In fact, Java does not contain pointers that can be explicitly manipulated (or abused) by the programmer. Although this problem may be nearly solved for Java development, it still remains a serious issue in legacy programming languages such as C. Since it is easier to write Java code that does not suffer from dynamic memory errors, it is logical to question the need to understand C programming techniques and tools. Although in some cases that do not require high performance, Java may be the correct choice, most legacy program products in the Unix industry have already been written in C, and will not be rewritten. For example, both the AIX and Linux operating systems have been written mostly in the C language. This paper proposes solutions to the challenging programming problems relating to

handling dynamically requested memory on the heap in the C language on AIX.

2.1 Invalid Memory Access

The largest and most common category of dynamic memory programming errors involves memory access. These all involve either attempts to write to or read from memory to which either the process can not, or should not be accessing. Of these two, invalid memory writes tend to be more damaging. However, invalid memory reads can also cause incorrect program behavior. A common example would be an attempt to either read or write to location zero, also known as de-referencing a null pointer. (e.g., `{char * p; p=0; *p='A';}`) However, a program which tries to modify location zero will immediately cause a core dump, making this form of error very easy to locate and fix, since the resulting core dump shows exactly the invalid reference is taking place. This paper will focus on more insidious and challenging programming errors that only show up indirectly and often in seemingly nondeterministic conditions.

A characteristic of these subtle dynamic memory bugs is demonstrated in that each of the invalid memory access example programs demonstrated in this paper will, by default, run fine on AIX, without causing any apparent problems. Although these programs do demonstrate real logical programming errors, they may appear to be irrelevant in that they do not immediately cause severe problems like core dumps. In a real world application program, the memory access patterns would be more interesting. They would modify memory further away from the accessible allocated memory, and would have the potential for more serious side effects. The important point is that while serious memory access programming errors may cause an immediate core dump, many invalid memory access errors do not show any symptom immediately. Yet they still reduce total software quality and reliability when they cause difficult to identify and fix core dumps or behavior problems in the future.

Perhaps the most common invalid memory access problem is a buffer overrun, in which a program accesses memory beyond the end of the buffer it is using. When a single buffer is memory returned from a single call to a `malloc()` function, with certain caveats, the tools introduced later in this paper can detect this invalid access fairly easily. The following program, `bufferoverrun.c`, demonstrates a common "Off By One Bug" (OBOB), where the program attempts to access the first byte following the allocated memory.

.

Program 1: `bufferoverrun.c`

```
/* malloc buffer overrun example: bufferoverrun.c */
#include <stdlib.h>
#include <stdio.h>

main()
{
   char *m;

   m=malloc(10);

   m[10]='A';      /* overrun - invalid memory write */
   putchar(m[10]); /* overrun - invalid memory read */

   free(m);
}
```

The underrun is similar to an overrun, except it accesses memory before the buffer rather than after the buffer. In addition to potentially overwriting critical program data, both overrun and underrun memory writes to memory surrounding a `malloc()`ed region are dangerous since

they can overwrite `malloc()` header information which is located before and after individually `malloc()`ed memory chunks. Without special debugging tools, this sort of error will not immediately result with a detectable problem, but rather eventually, subsequent calls to `malloc()` or `free()` may either core dump, or may end up returning spurious pointers which may further result in more invalid memory writes. This directly causes either incorrect program behavior, or a core dump. In this case, due to the cascade effect, the final core dump usually does not have any obvious indication of where the real problem occurred. It is common for programs with these kinds of programming errors to core dump in the AIX `libc.a` `malloc()`, `free()` functions, or in `libc.a` routines that are called by `malloc()`. At first glance with a problem like this, it appears that there is a bug in the AIX library, however, in Generally Available (GA) level AIX, this is always due to a misbehaving program rather than an error in AIX's `libc.a`.

Program 2: `bufferunderrun.c`

```
/* malloc buffer underrun example: bufferunderrun.c */
#include <stdlib.h>
#include <stdio.h>

main()
{
   char *m;

   m=malloc(10);

   m[-1]='A';      /* underrun - invalid memory write */
   putchar(m[-1]); /* underrun - invalid memory read */

   free(m);
}
```

A special case of invalid memory access in demonstrated below when freed memory is accessed. Besides being a logical programming error, writing to freed memory can cause a problem when this memory is reallocated in the future. The tools introduced later can also detect this sort of error before it causes a serious problem.

Program 3: `freedaccess.c`

```
/* demonstrate access to freed memory: freedaccess.c */

#include <stdlib.h>
#include <stdio.h>

main()
{
   char *m;

   m=malloc(10);

   free(m);

   m[5]='A';      /*  invalid memory write to freed memory */
   putchar(m[5]); /*  invalid memory read  to freed memory */
}
```

Another special case is the attempt to read memory that has not yet been initialized. Global variables and memory that has been reserved via `calloc()` are guaranteed to be initialized to zero. Although local variables and memory obtained with `malloc()`, `alloca()`, and `valloc()` are commonly zero on first read access, this behavior should not be depended on. `lint` will identify read access to uninitialized local variables, and a memory debug tool such as ZeroFault is needed to detect invalid access to uninitialized memory received from the malloc()

family of function calls.

Program 4: `uninitread.c`

```
/* uninitialized memory reads: uninitread.c */
#include <stdlib.h>

int global_variable;  /* auto initialized to zero */

main()
{
   int local_variable; /* not guaranteed to be initialized */
   char * m, * c;

   printf("global_variable=%d¥n",global_variable);
   printf("local_variable=%d¥n",local_variable);

   m=malloc(15); /* not guaranteed to be initialized */
   printf("malloc()ed byte=%d¥n",m[5]);

   c=calloc(15,1); /* auto initialized to zero */
   printf("calloc()ed byte=%d¥n",c[5]);

   free(m);free(c);
}
```

2.2 Dynamic Memory API Usage Errors

There are also several types of programming errors involving passing incorrect arguments to the various dynamic memory functions in `libc.a`. These errors can be easily identified with the automated tools introduced later.

The `free()` function is used to release memory that has been allocated with the `malloc()` family of functions. The argument passed to `free()` must be an address returned by an earlier `malloc()` family function. Passing in any other pointer, a previously freed address, an address outside of `malloc()`ed memory, or any other address that was not explicitly returned by `malloc()` family functions, is a programming error, and should be avoided. The following are some examples of invalid arguments passed to the `free()` function.

Program 5: `badfree.c`

```
/* demonstrate bad arguments to free(): badfree.c */

#include <stdlib.h>

main()
{
   char *m;

   m=malloc(10);

   free(0);         /* null pointer to free: defined to be ignored */
   free(0x1234567);/* random and invalid argument to free */
   free(m+5);      /* not the pointer returned by malloc() */
   free(m);         /* this is correct. */
   free(m);         /* oops.  Should not free twice! */
}
```

2.3 Memory Leaks

The final category of dynamic memory programming errors is known as a memory leak. A program with a memory leak is observed with commands like `ps` or `svmon` to use more and more memory. Systems running a program that has a memory leak will show increasing

paging space usage with commands like `lsps` and `topas`. When the offending program is terminated, this paging space will be released to the system. Sometimes true memory leaks are confused with programs that simply use a lot of memory, or have not yet reached a steady state with respect to working set size.

The following simple program demonstrates a memory leak. Each time a user enters a string, the program requests an additional 4K buffer from `malloc()`, which is never freed. In short lived processes memory leaks may not cause any problems, since all memory requested with the `malloc()` family is released to the system upon process termination. However, in a long lived process such as a daemon, a web server, the X server, a window manager, or most kinds of e-commerce server programs, even a small memory leak may have serious consequences over time.

Program 6: `memleak.c`

```
/*  sample memory leak: memleak.c */

#include <stdio.h>
#include <stdlib.h>

main()
{
   char * m;

   while (1) {
      m=malloc(4096);
      if (!fgets(m,4096,stdin))
          exit(0);
   }
}
```

3.0 Dynamic Memory Debug Tools

There are three primary kinds of dynamic memory debug tools currently available. These tools range from comprehensive commercially available development tools, to public domain replacements for `malloc()` and `free()` which include instrumented debugging code, to the user level dynamic memory debugging functionality that was added to AIX 4.3.3. This section will describe the pros and cons of the various approaches, along with complete and detailed usage examples, designed to assist a programmer putting these techniques to practical use.

The primary tools showcased in this section, ZeroFault, Electric Fence, and most importantly the built-in AIX debug malloc functionality have all been used with great success by the author in performing AIX Operating System development at IBM-Austin. Each of these tools has enabled the detection of otherwise hard to locate programming errors in a short time, and has thus contributed to improving the quality of the AIX Operating System. According to a search of the AIX CMVC defect database, ZeroFault tool has been instrumental in resolving 199 defects in IBM Development labs for both AIX and PSSP software between 1996 and 2001. A similar search revealed that Electric Fence has been used to resolve 47 defects between 1994 and 2001. ZeroFault and Electric fence are often used as debugging tools to resolve existing problems, where as the built-in AIX debug malloc is also used to **find** defects that would have otherwise been missed. The AIX debug malloc has successfully been used to find and fix 163 defects in a period of just over one-year starting in March 1999.

It is vital to use a relevant dynamic memory debugging tool during the various testing phases of software development in order to detect and remove these dormant errors before code is shipped into the field. This will result in improved software quality and reliability. Improved code

quality will also reduce maintenance expenses and reduce Customer Total Cost of Ownership (TCO). Additionally, if unusual, seemingly non-deterministic behavior is observed, often tools like this can quickly identify the cause of the problem, saving a programmer considerable time. Since a number of tools are available in AIX, a programmer has the option to select the correct tool for current testing or debugging requirements.

3.1 Commercial Dynamic Memory Debug Tools

There are a variety of commercially available tools that can be used on AIX for resolving dynamic memory programming errors. These include ZeroFault from the Kernel Group (TKG), Test Center from Centerline, Great Circle from Geodesic Systems, etc. (See the reference section for relevant URL's.) These tools tend to offer Graphical User Interfaces, comprehensive features to assist the programmer, and a non-trivial licensing fee.

In order to help the reader understand the usage and value of these tools, I will describe the tool ZeroFault from the Kernel Group (TKG), since it is a good representative. The Kernel Group is based in Austin Texas, and as an IBM Business Partner, they have been long involved with AIX development, and have made many direct contributions to improving the quality of the AIX operating system. The ZeroFault product is based on internal tools and technology that TKG used when working on AIX, and in fact, ZeroFault is only available on AIX.

Installation of ZeroFault can be accomplished using either "`smitty install`" or directly with the `installp` command to install the `ZeroFault.obj` fileset. After this fileset is installed, an individual user may need to set the following environment variables. (Note that `<licenseserver>` should be replaced with address of the relevant license server.)

Excerpt from `$HOME/.profile`

```
export TKG_LMHOST=@<licenseserver>
export ZF_HOME=/usr/lpp/ZeroFault
export PATH=$PATH:$ZF_HOME/bin
export ZF_SOURCE_PATH="/local/projects/src ."
```

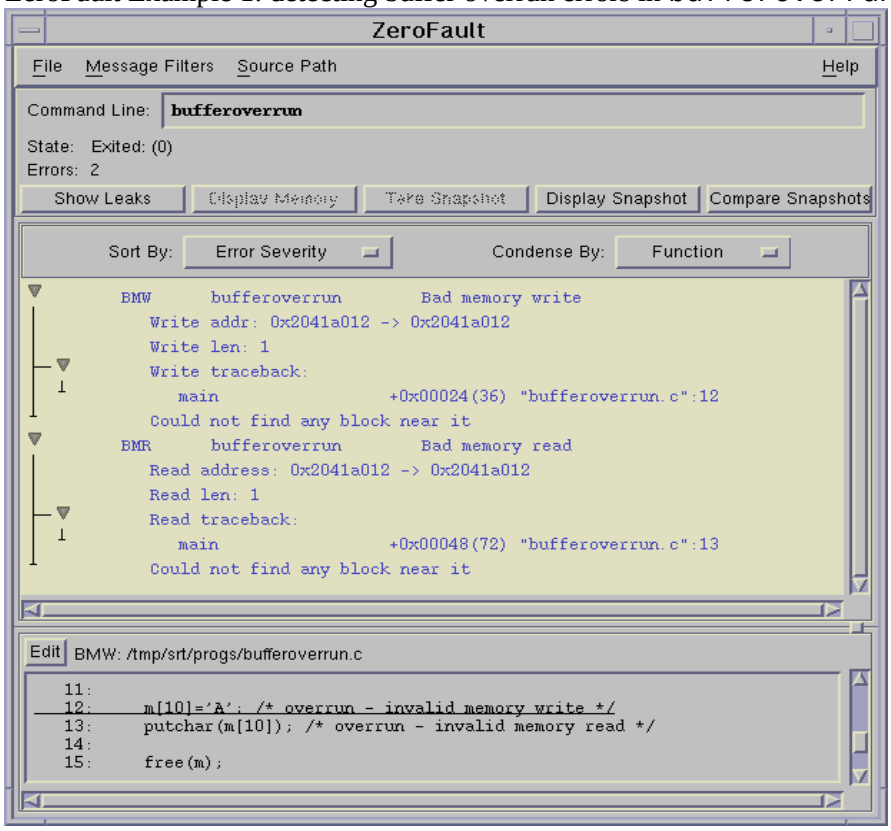
After ZeroFault is installed and the appropriate environment variables are set, invoking the utility is a trivial matter of prepending "`zf`" to the invocation of the command to be tested. Although it is easiest to debug programs compiled with the `-g` debug flag, and this technique was used for all ZeroFault examples in this section, compiling in debug mode is not an absolute requirement.

Compiling and running `bufferoverrun.c` under ZeroFault:

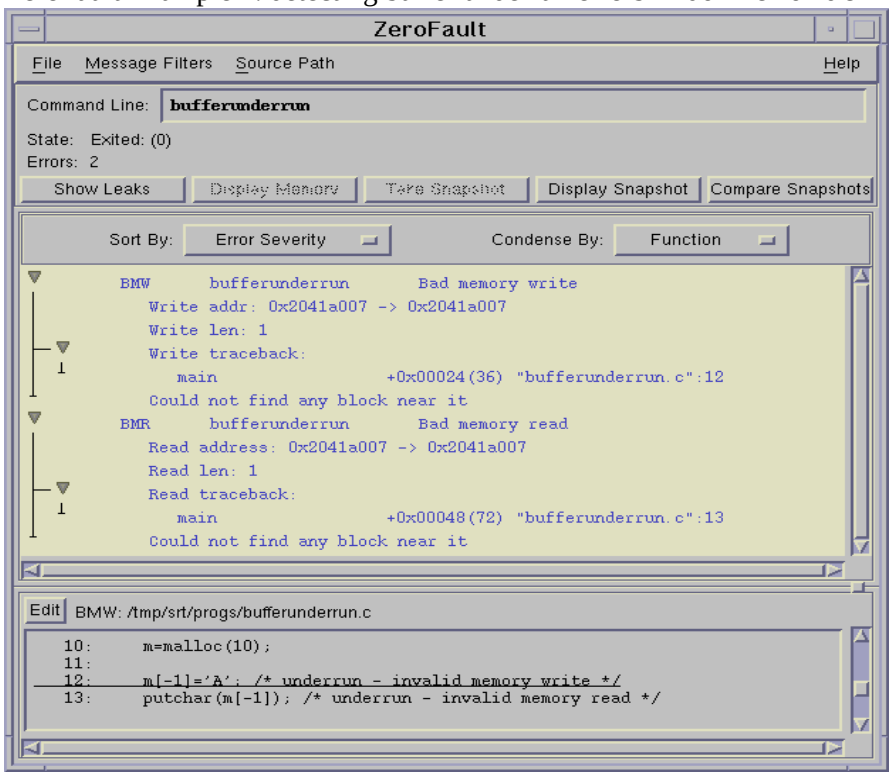
```
# cc -g bufferoverrun.c -o bufferoverrun
# zf bufferoverrun
```

The following two examples demonstrate how ZeroFault detects all invalid memory access errors in `bufferoverrun.c` and `bufferunderrun.c`. The ZeroFault user interface has an error window which describes the types of errors found, along with useful information for the programmer such as the arguments passed into relevant functions, and an identification of relevant source code lines. ZeroFault also displays a source code window where the currently selected source line from the error window (e.g., the offending source line) is highlighted with an underline. Unlike other tools, underrun errors can be detected without changing environment variables or settings. One of the significant advantages that ZeroFault has over other tools presented here, is that it will display all detected errors at the same time, where as the other tools will terminate after the first error is identified.

ZeroFault Example 1: detecting buffer overrun errors in `bufferoverrun.c`

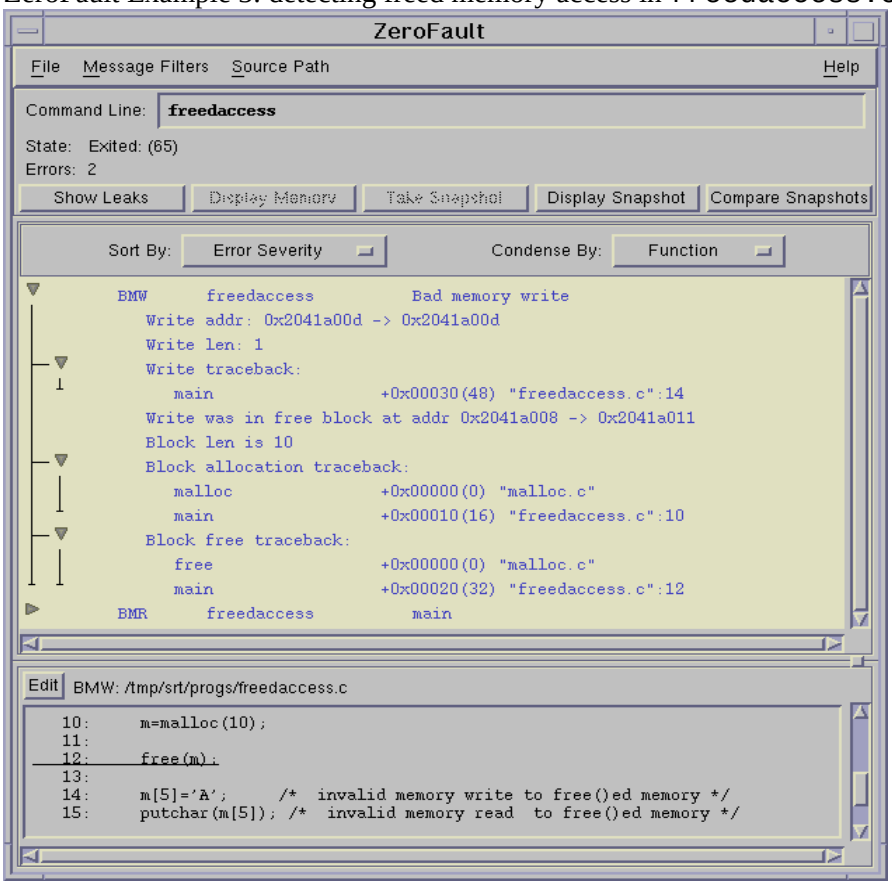


ZeroFault Example 2: detecting buffer underrun errors in `bufferunderrun.c`



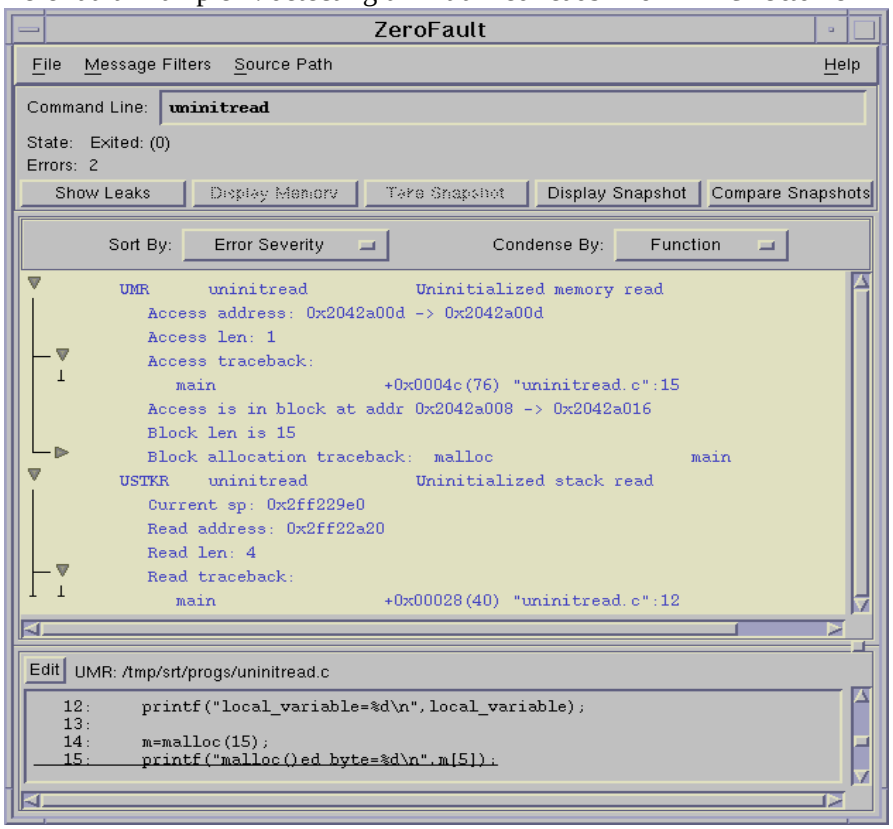
The output for Example 3 demonstrates invalid memory access to freed memory, and is very similar to the previous examples. As before, no user special action or configuration was needed to detect these errors. Notice that in addition to identifying the location of the invalid memory read or write, ZeroFault also identifies the location of the call to `free()` that released the region of memory that was used for the invalid access. This is useful since the programming error may not be the access, but may be in the fact that the memory was freed when it should not have been. It may also be worthwhile to point out that although ZeroFault is not an Integrated Development Environment (IDE), it is possible to use the GUI to directly edit the source code in order to fix source code defects.

ZeroFault Example 3: detecting freed memory access in `freedaccess.c`



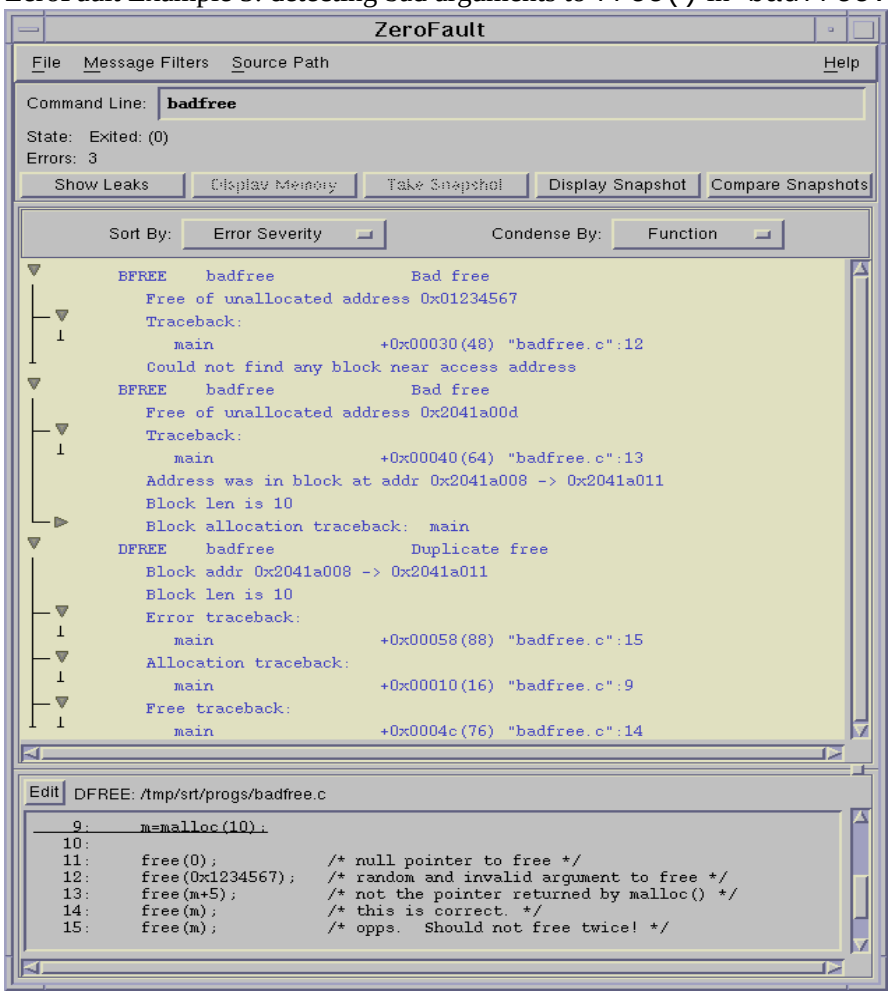
In the following example, ZeroFault detects a read to uninitialized `malloc()`ed memory as well as a read access to the uninitialized local variable, `local_variable`. Although invalid access to uninitialized memory may not be as catastrophic as other forms of dynamic memory errors, this is still an easily detectable and easily resolvable condition, which should not be ignored. Attempts to access global memory and `calloc()`ed memory which has not been explicitly initialized are not flagged, since as described elsewhere, these are implicitly initialized to zero by the system.

ZeroFault Example 4: detecting uninitialized reads in `uninitread.c`



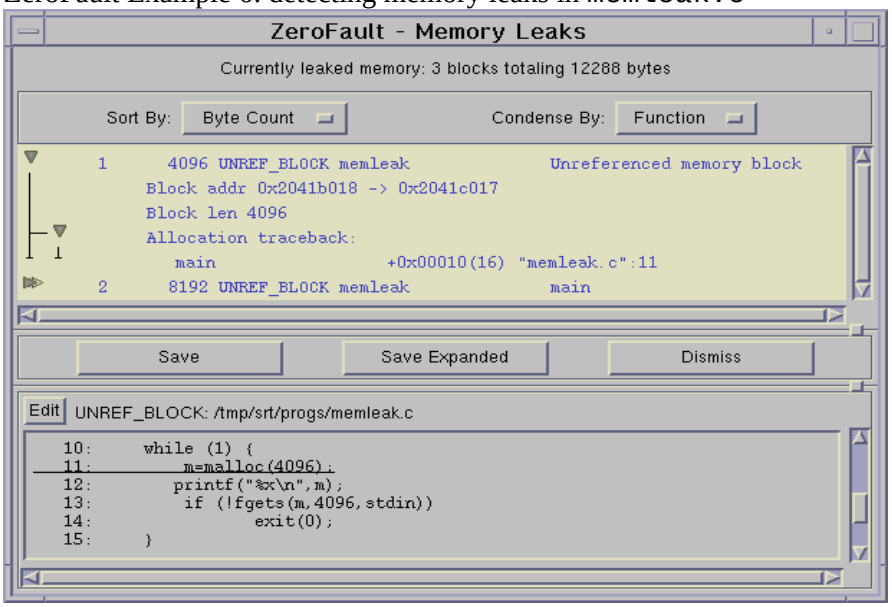
The following example shows a program that passes a variety of invalid arguments to the `free()` function. Since the `free()` function is defined to take no action when it is passed a null pointer, ZeroFault does not flag `free(0)` as an error. Obviously incorrect pointers passed to `free()` are all identified with the offending line in the program. The sample program demonstrates that calling `free()` twice on a valid pointer returned from `malloc()` will succeed the first time, but will generate a duplicate free error on subsequent calls. In addition to locating the erroneous duplicate `free()`, ZeroFault also indicates where the successful `free()` and the corresponding `malloc()` are located. It is beneficial to know which `malloc()` was involved since it helps the programmer understand the context of the memory use. It is useful to know the location of both `free()`s since even though the first one succeeded and the second one failed, it is possible that the first `free()` should not have been called and is in error.

ZeroFault Example 5: detecting bad arguments to `free()` in `badfree.c`



ZeroFault also keeps track of memory blocks that have no current references made to them. In other words, Zerofault can be used to identify memory leaks and the location in the source code where the lost memory was `malloc()`ed. In the example below, there were three unreferenced memory blocks (i.e., memory was "leaked" three times), at the time that the screen shot was taken. Detailed information about the first memory leak is displayed. Detailed information about subsequent leaks can be viewed by clicking on the twisty with the 2 next to it at the bottom of the error window. ZeroFault also has the ability to save a memory profile, and compare multiple memory profiles. This can be beneficial in understanding which memory segments are requested and which are lost in each iteration of an application.

ZeroFault Example 6: detecting memory leaks in `memleak.c`



ZeroFault has several merits, including the fact that it detected more errors than the other techniques demonstrated later in this paper, and did so without requiring any potentially confusing user configuration. The more complicated tool configuration is when attempting to detect programming errors, the more likely it is that these errors will be missed during the development process. ZeroFault can also save a programmer development time since it displays all errors with an intuitively configurable level of detail, unlike other techniques which require the test program to be rerun each time an error is removed. Since ZeroFault depends on a GUI and human analysis of the output, it may not be suitable for automated test suites. Finally, the following chart demonstrates the performance impact of using ZeroFault on a simple benchmark, `membench.c`, which is documented in the Appendix of this paper.

ZeroFault performance analysis

Environment	Performance Penalty (time to run `membench` with ZeroFault / time to run `membench` without ZeroFault)
test program compiled without `-g`	8.928
test program compiled with `-g`	10.302

3.2 Public Domain `malloc()`/`free()` Replacements

There are a variety of public domain packages which replace the standard `malloc()`, `free()`,

etc. routines with carefully written functions that help identify most of the dynamic programming memory problems discussed in Section 2. Representatives of this category of tools that work well on AIX include dmalloc – Debug Malloc Library, mpatrol debug library, and the Electric Fence. (See the reference section for relevant URL's.)

To help the reader understand the usage and considerations involved with this category of debugging tools, I will demonstrate the use of the Electric Fence utility to analyze the behavior of the buggy programs described in Section 2. Although each of these programs contains an error that can be identified with either close code inspection, or with the appropriate tool, by default, none of them will exhibit any obvious errors, such as core dumps.

Currently, the Electric Fence package, version 2.0.5 can be obtained from `http://sunsite.unc.edu/pub/Linux/devel/lang/c/ElectricFence-2.0.5.tar.gz`. Although the file is in a directory labeled "Linux", Electric Fence will build on AIX without any source modifications. The only thing that needs to be done is to edit the `Makefile`, and uncomment the line for AIX as described by comments in the `Makefile`. Then running `make` will successfully build `libefence.a`.

I placed the `libefence.a` object in the same directory with my test programs, and then used the following syntax to recompile each test program. (Relinking would also be sufficient.)

```
cc -bnso -bnodelcsect -bI:/lib/syscalls.exp -L. -lefence -g program.c -o
program
```

Program 1, `bufferoverrun.c` wrote one byte beyond the memory that was allocated with `malloc()`. This is a trivial example because `malloc()` will pad requested memory up to a power of two bytes, which means that unless you have requested an even power of two number of bytes, overwriting the requested memory buffer will often simply overwrite unused memory. This is still however, programming error that must be fixed, since future code changes could reduce the number of padding bytes (i.e., a request for 16 bytes would likely have less padding bytes than a request for 9 bytes). By default, Electric Fence on AIX will `malloc()` memory in units of 4 bytes, which means that, in our example, when we requested 10 bytes, we actually got 12 bytes, giving us 2 bytes worth of padding. Thus by default, Electric Fence will not detect the single byte buffer overrun in our example. Fortunately, Electric Fence provides the ability to tune the alignment behavior using the `EF_ALIGNMENT` environment variable. When this environment variable is set to "`0`", Electric Fence will detect this particular error as shown in the following example.

Electric Fence Example 1: detecting buffer overrun errors in `bufferoverrun.c`

```
# cc -bnso -bnodelcsect -bI:/lib/syscalls.exp -L. -lefence -g
bufferoverrun.c -o bufferoverrun
# bufferoverrun

  Electric Fence 2.0.5 Copyright (C) 1987-1995 Bruce Perens.
A

# export EF_ALIGNMENT=0
# bufferoverrun

  Electric Fence 2.0.5 Copyright (C) 1987-1995 Bruce Perens.
Memory fault(coredump)

# dbx bufferoverrun
Type 'help' for help.
reading symbolic information ...
[using memory image in core]
```

```
Segmentation fault in main at line 12 in file "bufferoverrun.c"
   12      m[10]='A';  /* overrun - invalid memory write */
(dbx)
```

Program 2, `bufferunderrun.c` wrote one byte before the beginning of the memory allocated by `malloc()`. By default, Electric Fence places a "guard page" after the allocated memory which will detect common overrun reads and writes. However, underruns such as demonstrated in this program can also occur. When we set the environment variable `EF_PROTECT_BELOW` to `1`, the guard page is placed before the allocated memory rather than after, and hence underrun errors will be detected.

Electric Fence Example 2: detecting buffer underrun errors in `bufferunderrun.c`

```
# export EF_ALIGNMENT=0
#  #  cc  -bnso  –bnodelcsect  -bI:/lib/syscalls.exp  -L.  -lefence  -g
bufferunderrun.c -o bufferunderrun
# bufferunderrun

  Electric Fence 2.0.5 Copyright (C) 1987-1995 Bruce Perens.
A

# export EF_PROTECT_BELOW=1
# bufferunderrun

  Electric Fence 2.0.5 Copyright (C) 1987-1995 Bruce Perens.
Memory fault(coredump)

# dbx bufferunderrun
Type 'help' for help.
reading symbolic information ...
[using memory image in core]

Segmentation fault in main at line 12 in file "bufferunderrun.c"
   12      m[-1]='A';  /* underrun - invalid memory write */
(dbx)
```

Program 3, `freedaccess.c` demonstrates that even though it is a logical programming error, by default, memory which has been freed, can still be accessed without generating an explicit error. This is also a condition that Electric Fence can detect easily, as shown below.

Electric Fence Example 3: detecting freed memory access in `freedaccess.c`

```
# cc -bnso –bnodelcsect –bI:/lib/syscalls.exp -L. -lefence -g freedaccess.c
-o freedaccess
# freedaccess

  Electric Fence 2.0.5 Copyright (C) 1987-1995 Bruce Perens.
Memory fault(coredump)

# dbx freedaccess
Type 'help' for help.
reading symbolic information ...
[using memory image in core]

Segmentation fault in main at line 14 in file "freedaccess.c"
   14      m[5]='A';     /*  invalid memory write to freed memory */
(dbx)
```

Program 4, `uninitread.c`, demonstrates reading uninitialized memory of various types. Since the Electric Fence will not detect any form of uninitialized memory read, I will skip the example, and go directly to the next type of error.

Program 5, `badfree.c`, demonstrates passing of various invalid arguments to the `free()`

function. Although Electric Fence will detect this error condition as demonstrated below, since the detection takes place dynamically within the `free()` function, error notification takes place through a deliberate illegal instruction fault caused within the `EF_Abort()` function. Sufficient information is provided to easily locate the offending line of source code, as demonstrated in the following example.

Electric Fence Example 4: detecting bad arguments to `free()` in `badfree.c`

```
# cc -bnso –bnodelcsect -bI:/lib/syscalls.exp -L. -lefence -g badfree.c
-o badfree
# badfree

  Electric Fence 2.0.5 Copyright (C) 1987-1995 Bruce Perens.

ElectricFence Aborting: free(1234567): address not from malloc().
Illegal instruction(coredump)

# dbx badfree
Type 'help' for help.
reading symbolic information ...
[using memory image in core]

Illegal instruction (reserved addressing fault) in EF_Abort at line 137
in file ""
couldn't read "print.c"
(dbx) where
EF_Abort(pattern = warning: Unable to access address 0x20001004 from core
(invalid char ptr (0x20001004)), 0x1234567, 0x2ff22ffc, 0xd030, 0x0,
0x60000000, 0x60000ecd, 0x0), line 137 in "print.c"
free(0x1234567), line 638 in "efence.c"
main(), line 12 in "badfree.c"
(dbx)
```

Since Electric Fence does not provide any facilities for locating memory leaks, no example on this topic is provided here.

The above examples demonstrate that Electric Fence can be easily used to determine the location of many forms invalid dynamic memory access such as `malloc()` buffer overruns, underruns, and freed memory access, as well as detect invalid arguments to `free()`. As compared with ZeroFault, one of the disadvantages of Electric is that it will only report a single error at a time. This may not be as inconvenient as it seems, since the first invalid memory operation will likely cascade causing subsequent invalid memory operations. The following chart demonstrates the performance impact of using Electric Fence on a simple benchmark, `membench.c`, which is documented in the Appendix of this paper. Notice that the `EF_PROTECT_FREE=1` option, which marks freed memory as invalid rather actually freeing the memory for system reuse, is an expensive operation.

Electric Fence performance analysis

Environment variable setting	Performance Penalty (time to run `membench` with Electric Fence / time to run `membench` without Electric Fence)
none	1.651
`EF_ALIGNMENT=0`	1.632
`EF_PROTECT_BELOW=1`	1.651
`EF_PROTECT_FREE=1`	3.828

3.3 AIX Debug Malloc Functionality

AIX has long provided the ability to use a debug kernel that will detect dynamic memory handling errors within the kernel. However, it was not until AIX Version 4.3.3 that functionality was added to AIX to enable it to detect dynamic memory handing errors within user level programs without the use of add-on products or libraries. This feature was originally added to improve the quality of AIX by making it easy to detect dormant bugs in AIX commands and libraries during internal development and test phases. According to a search of the Austin CMVC database of AIX defects, this functionality has enabled AIX development to locate and fix at least 163 different errors between January 1999 and March 2001. This is a real accomplishment since each of these defects was identified in development with the debug malloc facility before being reported by Customers in the field. Activities like this directly improve software quality, reliability and customer satisfaction, since these are defects that Customers will never see. This also results in a significant $2.5 Million USD savings for IBM since it is much less expensive to fix defects during the development phase ($25 per defect), than it is to fix defects found in the field by our Customers ($40K per defect). Due to its ease of use and implementation, and ability to detect otherwise hidden defects, each software development test phase should also include testing with the AIX debug malloc enabled. It is particularly well suited for use within automated test suites, as it can be fully utilized within a scripted environment.

There is no need to purchase or install additional software in order to utilize the debug malloc facility. Nor is it necessary to rebuild or relink programs or use a GUI. AIX debug malloc is enabled by simply setting the environment variables `MALLOCTYPE` and `MALLOCDEBUG` as described below.

First, debug malloc is enabled by setting environment variable `MALLOCTYPE` to the string "`debug`". Next, if desired, the environment variable `MALLOCDEBUG` can be set to specific values to alter the behavior of the debug malloc. The syntax for setting these options is `MALLOCDEBUG=[[ align:n | postfree_checking | validate_ptrs | override_signal_handling | allow_overreading | report_allocations | record_allocations],...]`.

In all of the examples that follow I will use "`align:0`" to ensure that even "Off By One Bugs" will be detected. Otherwise, as with the Electric Fence example, it is possible that either an overrun or an underrun of several bytes may be undetected as it may only access the padding bytes rather than causing a detectable access in a guard page. I will use the "`postfree_checking`" option to ensure that attempted accesses to freed memory are treated as errors. This option automatically sets "`validate_ptrs`" so that invalid arguments to `free()` will also be treated as errors. I will not set the "`override_signal_handling`" option, as this is only needed if running debug malloc on a program like `/usr/bin/bsh` which traps `SIGSEGV`. (Programs that trap SIGIOT should also use this option.) Nor will I set the "`allow_overreading`" option which prevents treating invalid reads as an error. (This could be useful to ignore the relatively unimportant invalid reads, and focus on invalid writes.) When I revisit memory leaks, I will set the "`report_allocations`" option which will implicitly set the "`record_allocations`" option, thus recording and reporting all `malloc()` allocations that have not been freed when the process exits.

Although the debug malloc facility is excellent for easily locating memory handing errors, to prevent unnecessary CPU and memory usage, and to prevent premature program failure due to

unexpected memory access errors, it is not recommended that this be turned on by default in production use. It should only be turned on when actively testing specific programs. For example, it should not be turned on system wide in `/etc/profile` or `/etc/environment`, nor should it be enabled when starting the X server. Each call to `malloc()` consumes significantly more memory when debug malloc is enabled.

Additionally, it is recommended that 32 bit programs which use a lot of memory be compiled with "`-bmaxdata:0x80000000`" and run with "`ulimit –d unlimited`" and "`ulimit –s unlimited`" set when testing with debug malloc. When testing is finished, the default values for `ulimit` and `–bmaxdata` should be restored..

A detailed description of the debug malloc functionality can be found in file `/usr/lpp/bos/README` on an AIX version 4.3.3 system (search for "`Debug Malloc`"). This description can also be found in the "Debug Malloc" section of Chapter 19 in *AIX 5L Version 5.1 General Programming Concepts: Writing and Debugging Programs.*

In the following example, I demonstrate how even though this program contains an obvious error, in normal circumstances, it will run to completion without causing any problems. Fortunately, we can easily detect the error by turning on debug malloc. However, like Electric Fence, even though a code path may contain multiple memory access errors, the program will terminate when the first one is encountered. In the following examples, I compile programs with the debug flag `–g`, although this is not a requirement to use debug malloc. In a later example, I show a little known technique that can be used to determine the exact source line of a core dump even with programs not compiled with `–g`.

Built-in AIX Function Example 1: detecting buffer overrun errors in `bufferoverrun.c`

```
# cc –g bufferoverrun.c –o bufferoverrun
# bufferoverrun
A

# export MALLOCTYPE=debug
# export MALLOCDEBUG=align:0,postfree_checking
# bufferoverrun
Memory fault(coredump)
# dbx bufferoverrun
Type 'help' for help.
reading symbolic information ...
[using memory image in core]

Segmentation fault in main at line 12
   12     m[10]='A';  /* overrun – invalid memory write */
(dbx)
```

I will skip the `malloc()` buffer underrun example because, unfortunately, the built-in debug malloc facility will not detect a typical `malloc()` buffer underrun. This is due to a design decision made in AIX Development, based on the fact that overruns occur much more frequently than underruns.

The following example demonstrates how the built-in debug malloc facility can be used to detect access to memory that has been freed. As with other examples, it also demonstrates how easy it is to determine the offending line of code.

Built-in AIX Function Example 2: detecting freed memory access in `freedaccess.c`

```
# cc -g freedaccess.c -o freedaccess
# freedaccess
```

```
A

# export MALLOCTYPE=debug
# export MALLOCDEBUG=align:0,postfree_checking
# freedaccess
Memory fault(coredump)
# dbx freedaccess
Type 'help' for help.
reading symbolic information ...
[using memory image in core]

Segmentation fault in main at line 14
   14      m[5]='A';     /*  invalid memory write to freed memory */
(dbx)
```

Sample program `uninitread.c` demonstrates a read access to a local variable, a global variable, `malloc()`ed memory, and `calloc()`ed memory, all of which are uninitialized. As described earlier, the local variable and `malloc()`ed memory accesses are logical programming errors. The built-in debug malloc facility will however not detect either of these accesses. It is possible, though, to use `lint`, a built in tool found in fileset `bos.adt`, to detect the uninitialized local variable read access, as demonstrated below.

Built-in AIX Tool Example 3: detecting uninitialized reads in `uninitread.c` with `lint`

```
# lint uninitread.c | grep -w "before set"
"uninitread.c", line 12: warning: local_variable may be used before set
```

Similar to the Electric Fence example, the built-in debug malloc code generates an I/O Trap from within the `free()` function rather than causing a `SIGSEGV` when free is passed a bad argument. Debugging information including the invalid argument passed to `free()` is displayed. It is also possible to use `dbx` to determine and view the offending line number using the "`where`" command to find the line number and the "`l`" command to view this line.

Built-in AIX Function Example 4: detecting bad arguments to `free()` in `badfree.c`

```
# cc –g badfree.c –o badfree
# badfree

# export MALLOCTYPE=debug
# export MALLOCDEBUG=align:0,postfree_checking
# badfree
Debug Malloc: free() called with pointer not allocated by malloc.
Malloc arena is 0x20001010 to 0x20011000, pointer passed in is 0x1234567
Abort(coredump)
# dbx badfree
Type 'help' for help.
reading symbolic information ...
[using memory image in core]

IOT/Abort trap in raise at 0xd01761a8
0xd01761a8 (raise+0x4c) 80410014       lwz   r2,0x14(r1)
(dbx) where
raise(??) at 0xd01761a8
abort() at 0xd016f8c8
do_debug_free(??) at 0xd0168994
main(), line 12 in "badfree.c"
(dbx) l 12
   12      free(0x1234567);   /* random and invalid argument to free */
(dbx)
```

The following example demonstrates how debug malloc can determine where memory leaks are taking place. Each time `malloc()` is called, a truncated stack trace (up to 6 call levels) and the pointer to the allocated memory are recorded. When the process terminates, a report is

displayed to standard output. In the list from the example below, we can tell that `memleak.c` performed one `malloc()` that was not freed by termination time. This allocation took place in `main()` and was labeled "`Allocation #1`". The first allocation is a memory region starting at `0x2000400`. The second allocation is used internally by `malloc()` when it called `atexit()` to register a request to run the memory leak report when the process terminates, and thus is not a memory leak, but rather a necessary artifact resulting from using `atexit()`. This report provides a programmer with a rough idea of where to focus on when debugging memory leaks.

Built-in AIX Function Example 5: detecting memory leaks in `memleak.c`

```
# cc –g memleak.c –o memleak
# memleak

# export MALLOCTYPE=debug
# export MALLOCDEBUG=align:0,postfree_checking,report_allocations
# ./memleak
Current allocation report:
   Allocation #1: 0x20004000
      Allocation traceback:
      0x20005024  __start
      0x20005028  main
      0x2000502C  malloc

   Allocation #2: 0x20001FF0
      Allocation traceback:
      0x2000201C  __start
      0x20002020  main
      0x20002024  malloc
      0x20002028  atexit
      0x2000202C  malloc

Total allocations: 2.
```

Although the examples above have all been compiled with the `–g` debug compile flag, it is possible to determine the exact location of an offending memory access without `–g` compiled code. In the example below, `bufferoverrun.c` is compiled without `–g`. However, we do specify `–qlist`, which will create the compiler listing `bufferoverrun.lst` shown below. This compiler listing can be created as a separate step if we have the same source code. Next we set `MALLOCDEBUG` and `MALLOCTYPE`, and run the program which will generate a core file. When we run `dbx` on the core file, since the program was not compiled in debug mode, we can not directly learn which source line made the invalid access. We can learn the offset from the current function, which in this case is `main+0x24`. We can now look at the `bufferoverrun.lst` compiler listing, and look for the line that has `0x24` in the second column (offset from start of function) for function `main()`. When we find this line, we then look at the original source line number in the first column, which is 12. Thus even though the offending program was not compiled with `–g`, we have learned that the invalid memory access took place on line 12 of `bufferoverrun.c`. If necessary, this technique will also work with Electric Fence, or with any executable for which we have either the source code or a current compiler listing.

Built-in AIX Function Example 6: debugging code without `–g` in `bufferoverrun.c`

```
# cc bufferoverrun.c -o bufferoverrun –qlist
# export MALLOCDEBUG=align:0,postfree_checking
# export MALLOCTYPE=debug
# bufferoverrun
Memory fault(coredump)

# dbx bufferoverrun
```

```
Type 'help' for help.
reading symbolic information ...warning: no source compiled with -g

[using memory image in core]

Segmentation fault in main at 0x1000033c
0x1000033c (main+0x24) 9864000a       stb   r3,0xa(r4)
(dbx)
```

`bufferoverrun.lst` excerpt, from `bufferoverrun.c` compilation with `–qlist`

```
     | 000000                          PDEF    main
   0|                                 PROC
    0| 000000 mfspr    7C0802A6   2    LFLR    gr0=lr
    0| 000004 stu      9421FFB0   1    ST4U    gr1,#stack(gr1,-80)=gr1
    0| 000008 st       90010058   1    ST4A    #stack(gr1,88)=gr0
   10| 00000C cal      3860000A   1    LI      gr3=10
   10| 000010 bl        4BFFFFF1   0     CALL     gr3=malloc,1,gr3,malloc
",malloc",gr1,cr[01567]",gr0",gr4"-gr12",fp0"-fp13"
   10| 000014 cror     4DEF7B82   1
   10| 000018 cal      38830000   1    LR      gr4=gr3
   10| 00001C st       90810040   1    ST4A    m(gr1,64)=gr4
   12| 000020 cal      38600041   1    LI      gr3=65
   12| 000024 stb      9864000A   1    ST1Z    (*)uchar(gr4,10)=gr3
```

The use of AIX's built-in debug malloc can be a very convenient and useful tool to detect invalid memory writes and reads when they first happen, rather than after they cause indirect and very hard to trace problems. This technique is convenient in that no modification to the target program is required, nor is it necessary to purchase or install new software. The fact that native debug malloc works in exactly the same environment is important since the behavior of dynamic memory errors can change or even vanish when the environment changes. This tool is also well suited for use within automated test suites. The following chart demonstrates the performance impact of using AIX debug malloc on the simple benchmark, `membench.c` which is documented in the Appendix of this paper. This solution has the best performance of the tools showcased in this paper.

Built-in AIX debug malloc performance analysis

`MALLOCDEBUG` value	Performance Penalty (time to run `membench` with debug / time to run `membench` without debug)
none	1.055
`align:0`	1.079
`align:0,postfree_checking`	1.507
`align:0,postfree_checking,report_allocations`	1.675

4.0 Conclusion – The Rewards

The first step to overcoming any problem is to understand that problem. Software developers who understand the principals introduced in this paper have taken the first step to reducing their overtime and producing higher quality code. Companies with developers who utilize these techniques will produce higher quality software, quicker. With the cost of fixing defects in the field as high as $40,000 per defect as compared to $25 for defects fixed in the development phase, these companies will save money. And with Customer system downtime costing as much as six million dollars per hour, fewer defects found in the field will also result in customer savings improved Customer satisfaction.

5.0 References

Electric Fence	sunsite.unc.edu/pub/Linux/devel/lang/c/ElectricFence-2.0.5.tar.gz
Dmalloc library	www.dmalloc.com
Memory Patrol	www.cbmamiga.demon.co.uk/mpatrol
ZeroFault	www.tkg.com/zf
Centerline	www.centerline.com/productline/test_center/test_center.html
Great Circle	www.geodesic.com/solutions/greatcircle.html
AIX on-line Docs	www.rs6000.ibm.com/cgi-bin/ds_form
AIX malloc debug	www.rs6000.ibm.com/doc_link/en_US/a_doc_lib/aixgen/relnotes/10069706.htm#Header_45
IBM Redbooks	www.redbooks.ibm.com
UW-Madison software reliability research	www.cs.wisc.edu/~bart/fuzz/fuzz.html
CMU software reliability research	www.cs.cmu.edu/afs/cs.cmu.edu/project/edrc-ballista/www

Appendix A: `membench.c`

`membench.c` is a simple but original test program that makes intensive use of `malloc()` and `free()` and read/write access of associated memory in order to measure the performance penalty of the various dynamic memory debugging tools showcased in this paper.

`membench.c`

```
#include <stdlib.h>

#define BUFFER_SIZE 2048
#define ITERATIONS 25000

void foo(int i)
{
   char *c;
   int j;

   c=malloc(BUFFER_SIZE);
   for (j=0;j<BUFFER_SIZE;j++) /* write to memory */
      c[j]=(char)i;
   for (j=0;j<BUFFER_SIZE;j++) /* read from memory */
      i=c[j];
   free(c);
}

main()
{
   int i;

   for (i=0;i<ITERATIONS;i++)
      foo(i);
}
```