

**AmigaMail**

<b>COLLABORATORS</b>
----------------------

	<i>TITLE :</i> AmigaMail		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY		July 23, 2024	

<b>REVISION HISTORY</b>
-------------------------

NUMBER	DATE	DESCRIPTION	NAME

# Contents

<b>1</b>	<b>AmigaMail</b>	<b>1</b>
1.1	I-47: Debugging with Enforcer and Mungwall . . . . .	1
1.2	The Debugging Setup . . . . .	3
1.3	Sample Enforcer Output . . . . .	3
1.4	Sample Mungwall Output . . . . .	5
1.5	Using Enforcer and Mungwall Together . . . . .	6
1.6	A Sample Debugging Session . . . . .	7
1.7	More Remote Debugging Tips . . . . .	8
1.8	Who Should Use Enforcer and Mungwall . . . . .	9

## Chapter 1

# AmigaMail

### 1.1 I-47: Debugging with Enforcer and Mungwall

by Carolyn Scheppner

It is almost impossible for a human being to develop software without including some bugs. Some bugs make themselves known rather quickly. Other bugs are not so easy to notice. For example, two hard to find bugs that sometimes slip past developers are using uninitialized pointers and using memory which has already been freed. These bugs often reference semi-random memory, which contains semi-random data. Typically the behavior of the software depends on the data passed to it. If that data is erratic, the behavior of the software will also be erratic. Because the behavior is erratic and unpredictable, the problem is very hard to spot. Quite often, bugs like these go unnoticed until software is running in a real user's multitasking environment. Fortunately, debugging tools like Enforcer and Mungwall (and for assembler programmers, Scratch by Bill Hawes) help uncover hidden bugs.

Enforcer is a debugging tool written by Bryce Nesbitt. Its job is to report any attempts to access regions of the Amiga address space that are off-limits to applications. These off-limits accesses include reading and writing to the lowest 1K range of memory (except for reading ExecBase from \$4), writing to the Kickstart ROM, and reading and writing to non-existent memory ranges.

Enforcer requires that the CPU has access to an MMU (Memory Management Unit) to catch reads and writes to memory. Most 68020 based accelerator boards have MMU chips on them. An MMU is built-in to each 68030 and 68040 CPU (currently Enforcer does not support 68040).

When an application tries to read a memory location that has been read-protected by Enforcer, Enforcer intercepts that memory read, reports the illegal memory access (known as an ``Enforcer hit''), and shows the application a zero instead of the contents of the memory address. The application has no idea that Enforcer did anything. When an application tries to write to a memory location that has been write-protected by Enforcer, Enforcer prevents the illegal memory write and issues an Enforcer hit.

---

Enforcer is even more powerful when used in combination with Mungwall. Mungwall was written by Ewout Walraven and is based on Memmung by Bryce Nesbitt and Memwall by Randell Jesup. The ``mung'' part of Mungwall fills all of free memory (and all subsequently freed memory) with a large, odd, 32-bit value. An odd value is likely to cause serious problems for any program that uses wild or uninitialized pointers, or uses memory after it has been freed.

Unlike Enforcer, Mungwall does not require any special hardware. Mungwall can run without Enforcer and on non-MMU machines.

Mungwall uses several special 32-bit values to ``mung'' memory which helps diagnose problems:

Except when Enforcer is running, Mungwall sets location zero to \$CODEDBAD. Normally, location zero is \$00000000. By putting an odd, non-zero value in location zero, any erroneous references to location zero are much more likely to show themselves. For example, a program that references location zero as character array will see a string that starts with the ASCII values \$C0 \$DE \$DB \$AD, rather than seeing a NULL string.

When Mungwall starts up, it sets all free memory to \$ABADCAFE. If this number shows up while an application is running, it is likely that someone is referencing memory in the free list.

When a program allocates memory, Mungwall sets that memory to \$DEADF00D (Except when allocating memory with MEMF\_CLEAR). When an application accidentally accesses its memory before initializing it, the application will find the well-known value \$DEADF00D rather than some random value that happened to be left in memory.

Mungwall fills deallocated memory with \$DEADBEEF, which makes using freed memory bugs much more obvious.

The ``wall'' part of Mungwall allocates extra memory before and after every memory allocation and fills this memory ``wall'' with a fill pattern (normally \$BB) and some information Mungwall uses to perform certain tests on an application's memory blocks:

When an application deallocates memory, Mungwall reports when the size of the deallocated memory block does not match its size when it was allocated.

Mungwall reports when a memory block's ``walls'' have been overwritten.

Mungwall reports allocations and deallocations of memory blocks that are zero bytes long and deallocations of memory blocks that start at location zero.

Mungwall has an option to ``snoop'' and report on all memory allocations and deallocations for all tasks or specific tasks. This feature can be useful when tracking down memory losses. Because the snoop option can generate so many reports, the output can be run through the snoopstrip program which will throw away all matching allocate/deallocate pairs.

---

The Debugging Setup  
 Sample Enforcer Output  
 Sample Mungwall Output  
 Using Enforcer and Mungwall Together  
 A Sample Debugging Session  
 More Remote Debugging Tips  
 Who Should Use Enforcer and Mungwall

## 1.2 The Debugging Setup

Enforcer and Mungwall both output their debugging information to the serial port at the baud rate to which the Amiga's serial hardware is currently set. After powerup, the serial hardware is set to 9600 baud. You can use a terminal package to alter the current baud rate setting. To set up an Amiga for serial debugging, connect your Amiga's serial port via a NULL-modem serial cable to a terminal.

The best debugging setup is to connect your Amiga to another computer running a terminal package. Ideally the terminal package should have an ASCII capture mode so it can capture all the serial debugging output and save it to a file for examination. The terminal package should also beep when it receives a CTRL-G, as both Enforcer and Mungwall send a CTRL-G beep with each Enforcer/Mungwall hit. There are several less effective alternatives to using Enforcer and Mungwall with a remote terminal. Both can send their output to a serial printer. There are special versions of both Enforcer and Mungwall that send their output to the parallel port (Enforcer.par and Mungwall.par) for output to a parallel printer. In a pinch, attach a modem to the serial port and run a terminal package set to the modem's baud rate. As long as the modem has not made any telephone connection, the modem will bounce back any Enforcer or Mungwall hits that come through the serial port, which the terminal package can capture. Because the debugging information has to move at modem's baud rate, the modem method tends to lose data, especially when there is a lot Enforcer/Mungwall hits.

## 1.3 Sample Enforcer Output

Here is a sample Enforcer hit which was caused by a program called Lawbreaker which tried to read from location \$14.

```

Program Counter (approximate)= 343F4A          Fault address      = ←
      14
User stack pointer           = 348734          DOS process address = ←
      339590
Data: DDDD0000 DDDD1100 DDDD2200 DDDD3300 DDDD4400 DDDD5500 DDDD6600 DDDD7700
Addr: AAAA0000 AAAA1100 AAAA2200 AAAA3300 AAAA4400 AAAA5500 AAAA6600 00002E28
Stck: 00210D70 00000FA0 00339F84 BBBBBBBB BBBBBBBB BBBBBBBB BBBBBBBB BBBBBBBB
READ-WORD  (---) (-) (-)      SR=0008      SSW=0161
Background CLI, "lawbreaker", Hunk #0, Offset $5A
  
```

Program Counter - Normally, this is the address of the instruction that was executing when the Enforcer hit occurred. For some types of

Enforcer hits, this is the address of the instruction that executed after the hit. Note that if a program passes a bad pointer or an improperly initialized structure to a system ROM routine, it can cause the ROM code to read or write to an illegal address.

Fault Address - This is the address where the illegal access occurred. In this example, the illegal access occurred at address \$14, and as specified later in the debugging output, this access was a READ-WORD access. So the illegal memory access was an attempt to read a word (2 bytes) at address \$14. Low memory accesses are often caused by NULL pointers to structures. If, for example, a ROM routine references a WORD-sized structure member at offset \$20, and an application passes the ROM routine a NULL pointer as a pointer to that structure, Enforcer will report that the ROM routine tried to read a word at address \$20.

User Stack Pointer - This is the value that was in register A7 when the Enforcer hit occurred. It points to the top of the stack for the task that was running when the Enforcer hit occurred.

DOS Process Address - This points to the Task structure of the task that was running when the Enforcer hit occurred.

Data/Addr (Register Dump) - This is the contents of registers D0-D7 and A0-A7. This information can help assembly programmers and programmers who like to debug at a low level. Notice that register A7, the stack pointer, no longer contains the stack pointer for the task that caused the Enforcer hit (A7 does not match the user stack pointer above). Ignore the value in A7.

Stck (Stack Dump) - This is the eight long words at the top of the offending program's stack. For those who need to see more of the stack, there is a special version of Enforcer called Enforcer.megastack. that shows the last 32 long words on the stack.

Access Type - This tells what kind of memory access the Enforcer hit is. In this example the access is a READ-WORD, which most likely is a result of Lawbreaker accessing a word-sized structure member. There are two other read access types: READ-BYTE, which is generally a result of a bad string pointer, and READ-LONG, which is normally a result of a bad pointer or a bad pointer within a structure. When an errant program directly or indirectly writes over Enforcer-protected memory, Enforcer reports a WRITE-BYTE, WRITE-WORD, or WRITE-LONG type access. When Enforcer issues an INSTRUCTION type memory access, the CPU tried to load an instruction from an invalid address. Some common causes for this type of Enforcer hit are:

An errant program has overwritten a program's instructions,  
An errant program has overwritten a subroutine return address  
that was on the stack,  
The CPU tried to execute a library function using an invalid library base.

Notice that the errant program is not necessarily the program that caused the Enforcer hit.

Interrupt/Forbid/Disable - The series of parentheses after the access type indicate if the Enforcer hit occurred in an interrupt (and if so, the interrupt level), Forbid, or Disable state:

---

(I-n) (-) (-)        The hit occurred in an n level interrupt.  
(---) (F) (-)        The hit occurred while the Amiga was in a Forbid state.  
(---) (-) (D)        The hit occurred while the Amiga was in a Disable state.

SR (Status Register) - The CPU's status register (see 680x0 manual for more information).

SSW (Special Status Word) - The special status word for 68010 though 68040 CPUs (see 680x0 manual for more information).

Program Name - The program name is the name of the task or command that was executing when the Enforcer hit occurred.

Hunk Number and Offset - If possible, Enforcer also provides a hunk offset to where the program counter was if the hit occurred within the program's own code instructions. This will only work if the errant program was started from a shell.

## 1.4 Sample Mungwall Output

When Mungwall reports a hit, it lists:

The function that triggered the hit (either AllocMem() or FreeMem()). It includes the functions arguments.

The name of the offending program (''attempted by '<program name>' '').

The address of the offending program's Task structure (''at 0x<task address>'').

The address from which AllocMem()/FreeMem() was called. Mungwall reports two addresses: one labelled ''A:'' and one labelled ''C: ''. The ''A:'' address is the address if AllocMem()/FreeMem() was called directly (i.e., using assembler or #pragmas) by the offending program. The ''C: '' address is the address if AllocMem()/FreeMem() was called from an amiga.lib C stub by the offending program. Since Mungwall patches the memory allocation functions, it can only guess the caller's address based on the return address it finds on the stack.

The stack pointer at the time of the Mungwall hit. It is labelled ''SP: ''.

Note that Mungwall ignores the layers.library's partial deallocations. If any other debugging tools patch AllocMem() or FreeMem(), Mungwall's ''A:'' and ''C: '' addresses may be thrown off by additional information pushed on the stack, and Mungwall will also be unable to screen out the layers.library's partial deallocations (which will often show up as Mungwall hits on your task, CON:, or input.device).

Here are some sample Mungwall hits that were produced by a program called mungwalltest. As a reminder, the arguments for memory functions are AllocMem(size,type) and FreeMem(address,size).

AllocMem(0x0,10000) attempted by 'mungwalltest' (at 0x339590)

---



```

from A:0x35C03A C:0x35677E SP:0x35CFC0      Tried to allocate 0 bytes
                                              of memory.

FreeMem(0x0,16) attempted by 'mungwalltest' (at 0x339590)
from A:0x35C068 C:0x3567C4 SP:0x35CFB8      Tried to free memory with
                                              a NULL pointer.

FreeMem(0x33BD10,0) attempted by 'mungwalltest' (at 0x339590)
from A:0x35C068 C:0x3567D4 SP:0x35CFB0      Tried to free 0 bytes of
                                              memory.

Mis-aligned FreeMem(0x33BD14,16) attempted by 'mungwalltest' (at 0x339590)
from A:0x35C068 C:0x3567E2 SP:0x35CFA8      Deallocation address is
                                              incorrect because it is
                                              not aligned according to
                                              AllocMem()'s lowest
                                              memory chunk size.

Mismatched FreeMem size 14!
Original allocation: 16 bytes from A:0x35C03A C:0x3567A0 Task 0x339590
Testing with original size.                  Deallocation size does
                                              not match the allocation
                                              size.

19 byte(s) before allocation at 0x33BD10, size 16 were hit!
>$: BBBBBBBBBB BBBBBBBBBB BB536572 6765616E 74277320 50657070 65722000
                                              Something wrote to the
                                              bytes which precede the
                                              allocation.

8 byte(s) after allocation at 0x33BD10, size 16 were hit!
>$: 75622042 616E6400 BBBBBBBBBB BBBBBBBBBB BBBBBBBBBB BBBBBBBBBB BBBBBBBBBB
                                              Something wrote to the
                                              bytes which follow the
                                              allocation.

```

## 1.5 Using Enforcer and Mungwall Together

Alone, Mungwall can catch a large variety of memory-related software problems. If Mungwall and Enforcer are used together, they can catch many more well-hidden bugs. One bug that is hard to catch is when a program either mistakenly reads memory that does not belong to the program or reads its own memory without initializing the memory first.

These bugs are normally hard to catch because the behavior of the errant program usually depends on how it reacts to the data that happens to be stored in that memory. This makes the behavior of the bug erratic. Sometimes the errant program may crash. Sometimes the errant program may write over another program's data, causing the other program to crash. At other times, no noticeable abnormality takes place.

Mungwall and Enforcer together can help find these bugs. Because Mungwall fills freed memory with the same odd 32-bit values, when an errant program mistakenly accesses memory, the behavior of the bug

will be consistent while Mungwall is running. Also, the values Mungwall puts in memory are more likely to cause the errant program to access Enforcer-protected memory, triggering an Enforcer hit.

## 1.6 A Sample Debugging Session

The following is an example of debugging an Enforcer hit that occurred using a test program called ownertest. This hit was generated on an A2500 with a 2.04 ROM image loaded using ZKick:

```
Program Counter (approximate)= 201946          Fault address      =      0
User stack pointer           = 566110          DOS process address = 38E888
Data: 00282D90 00000000 000003ED 0038FD8C 00000001 00000001 000E203B 00000001
Addr: 00225469 00000001 00282DE0 00448A3A 004487C0 004487CC 00001420 00002E28
Stck: 00448A3A 00223BA2 00280810 000003ED 0038FD8C 00000001 00000001 000E203B
READ-BYTE (---) (F) (-)      SR=0010      SSW=0751
Background CLI, "ownertest"
```

The Program Counter is at \$201946. On a machine with the Kickstart in ROM, the ROM addresses range from \$F80000 to \$FFFFFF under 2.0 or greater and from \$FC0000 to \$FFFFFF under 1.3. On a softkicked A3000, the addresses are in the same range as real ROM addresses. On a softkicked A2500, \$201946 is a ROM address (in this case, the ROM ranges from \$200000 to \$27FFFF). The first thing to do is figure out in which ROM module the Enforcer hit occurred. The debugging tool owner, by Michael Sinz, will figure that out:

```
1.Ram Disk:> owner 0x201946

Address  - Owner
-----  -
00201946 - in resident module: exec 37.132 (23.5.91)
```

Note that Owner looks at the ROM addresses of the Amiga on which it is executing, so you must run owner on the machine that generated the Enforcer hit.

Next, use the debugging tool lvo to figure out what function entry in the Exec ROM module is closest to that ROM address. Like owner, lvo also looks at local machine's ROM addresses, so you have to run owner on the machine that generated the Enforcer hit. Note that lvo requires the FD files to be in an ``FD:`` assign directory. Pass the address and the module name (from owner) to lvo.

```
1.Ram Disk:> lvo exec romaddress=0x201946

Closest to $201946 without going over:
exec.library LVO $feec -276 FindName() jumps to $20192a on this system
```

Hmmm. A lot of functions use FindName() on Exec lists, so, in this case, the Program Counter does not pinpoint the problem. However, it does hint that FindName() probably received a bad string pointer. The ``READ-BYTE`` attribute of the Enforcer hit is an extra clue that the Ownertest program has a problem with a string pointer, since the most common READ-BYTE actions are on strings.

Let's check the Enforcer stack dump and see if there are any other ROM addresses there which might have called FindName():

```
Stck: 00448A3A 00223BA2 00280810 000003ED 0038FD8C 00000001 00000001
      000E203B
```

The address closest to the top of the stack that looks like a ROM address on this particular system is 0x00223BA2. Let's see what ROM module contains this address:

```
1.Ram Disk:> owner 0x223BA2

Address  - Owner
-----  -
00223BA2 - in resident module: graphics 37.35 (23.5.91)
```

Let's see what function in the Graphics library is closest to this ROM address:

```
1.Ram Disk:> lvo graphics 0x223BA2

Closest to $223ba2 without going over:
graphics.library LVO $ffb8 -72 OpenFont() jumps to $223b84 on this
system
```

It looks like the ownertest program has a problem with a font name that was not properly initialized.

## 1.7 More Remote Debugging Tips

Debugging with Enforcer and Mungwall is even more effective when an application sends other debugging information to the serial or parallel port. The linker library debug.lib contains a printf()-like function, kprintf(), to print information to the serial port. The linker library ddebug.lib contains a similar function called dprintf() that prints debugging information to the parallel port. The output from these functions intermix with the output from Enforcer and Mungwall, making it easy to pinpoint which part of the code is causing Enforcer or Mungwall hits.

Functions like kprintf() and dprintf() are useful, but adding and removing them from programs can be tedious. One easy way to deal with this problem is to include them only when a special label is defined:

```
/******      debug macros      *****/
#define MYDEBUG 1
void kprintf(UBYTE *fmt,...);
void dprintf(UBYTE *fmt,...);
#define DEBTIME 0
#define bug printf
#ifdef MYDEBUG
#define D(x) (x); if(DEBTIME>0) Delay(DEBTIME);
#else
#define D(x) ;
#endif
```

```
#endif /* MYDEBUG */  
/***** end of debug macros *****/
```

Set MYDEBUG to 1 to turn on debugging. Set ``bug'' to:  
``printf'' to send debugging information to the default console,  
``kprintf'' to send debugging information to the serial port  
(link with debug.lib), and  
``dprintf'' to send debugging information to the parallel port  
(link with ddebug.lib).

When using this macro, make sure there two close parentheses before the semicolon at the end of each D(bug()) statement.

Example macro usage:

```
win = OpenWindow(&mynewwin);  
D(bug("Opened window at %lx\n", win));
```

A different low-level method of figuring out which instructions caused an Enforcer hit is to disassemble program memory where the hit occurred. First, match the disassembly with your own code. Assembly programmers could just compare the disassembly to their source. Others could take the hex values of a sequence of position-independent 68000 instructions near the hit (i.e. no addresses except for offsets and branches) and do a search for this pattern in your object modules. If you find the pattern, do a mixed source and object disassembly of that object module and then look in the output for instructions matching those where the hit occurred. For example, with SAS's OMD you could compile your code with the flag -dl, then do the following:

```
1.Ram Disk:> OMD >ram:dump mymodule.o mymodule.c
```

## 1.8 Who Should Use Enforcer and Mungwall

If you are developing Amiga software, it is extremely important that you invest in a MMU, or at the very least make sure that your software is tested on machines with Enforcer and Mungwall (also test with Enforcer alone as Mungwall can hide certain types of bugs). If you are programming in assembly, you should also test with Scratch by Bill Hawes to catch improper usage of CPU registers.

Enforcer and Mungwall are not just for developers and QA departments. Anyone who uses software can help find bugs in it with Enforcer. During normal usage, they can catch hidden software problems. Many people at Commodore run Enforcer all of the time. As more and more people begin running these tools, they will become less tolerant of software that causes Enforcer and Mungwall hits.

At a small developer meeting at a recent Amiga trade show, CATS was disappointed to discover that, although the majority of the audience believed that they needed Enforcer, a relatively small percentage of them owned the equipment necessary to run it (i.e., an MMU). If you don't have an MMU, get one. The investment in an A3000, 68030 card, or 68020+MMU card will quickly pay for itself. It significantly cuts down development time because it quickly catches bugs that are otherwise

---

hard to track down.