

Intro

It's been quite a few years since I did my tutorial for cracking Golden Axe. After recently releasing a javascript based Atari ST emulator I feel some nostalgia for the old 16 bit days so I figured it might be time to brush up on my old Amiga cracking skills and put something together for Flashtro.

I thought I'd start off with a game that we are all probably familiar with. Lemmings (2 disk version CAPS release 0132). I would classify this one as a medium to tricky one to crack. This tutorial is going to attempt to explain the steps in pretty basic terms but it's going to be a long slog and will require some basic understanding of the use of Action Replay and some understanding of MFM and load routines. This may not be the best tutorial for beginners. There's a lot of swapping disks and resetting and starting again from the bootblock code involved in following this tutorial. I won't be reminding you every time which disk needs to be in the drive, you must remember each time to swap disks when you need to load files into memory and swap back to the original or crack disks as needed.

I no longer own a real Amiga so for the purposes of this tutorial I'll be using WinUAE. For this game I'll be running using a standard A500 machine but to make things easier I like to take advantage of the following:

- Action Replay III Cartridge – useful for general poking around, reading disk data into RAM and messing with it. Good for assembling very small bits of code to test.
- Devpac – my assembler of choice.
- 2MB Fast RAM highly useful for dealing with large disk image data in memory with Action Replay. Located at \$200000 - \$400000 on my setup.

Our usual approach to cracking Amiga games would be to consider the following possibilities we may have to deal with:

- Non-standard MFM disk format(s)
- Single track protections (eg. Rob Northen CopyLock)
- Manual / Codewheel type protections
- Checksums
- Dongles (rare)

So our first step is to determine which of these we may have to deal with. Simply load up your lemmings disks and use your preferred copy program to look at the contents. You will see that only track 0 is standard and everything else is not. So now we know we have non-standard MFM disk format to take care of first. (See figure 1)

Investigation

So we start at the boot sector. Boot up till the insert disk screen appears then enter action replay insert the first disk and use the RT command to load the first track into memory (see figure 2).

Now let's disassemble the bootblock (see figure 3a and figure 3b).

What we see in the bootblock is some pretty standard stuff. First off is a call to -\$1c8 in exec.library – this should be very familiar. It's loading data from sector 2 onwards for \$1200 bytes (which would be the remaining 9 sectors of the first track). This is loaded at address \$70000. It then attempts to allocate memory in various configurations and stores the results in d0 and d1. It then jumps to \$70000 and starts executing the previously loaded code.

We already have the whole of track 1 loaded into memory at \$40000, so we can quickly copy the data to \$70000 (using the TRANS command) and see what is happening there. (See figure 4a, 4b and 4c).



Figure 1 - burst nibbler – built into action replay III

```

*****
ACTION REPLAY AMIGA MK III
(c) 1990/1991 by Olaf Boehm & Jörg Zanger
(p) by Datel Electronics Ltd
*****
No known virus in memory!
Ready.
rt 0 i 40000
Disk ok
-

```

Figure 2 – action replay- load track 0 into memory

```

d 4000c
~04000C MOVEA.L 00000004,A6
~040012 MOVEA.L #70000,A5
~040018 MOVE.W #2,1C(A1)
~04001E MOVE.L A5,28(A1)
~040022 MOVE.L #400,2C(A1)
~04002A MOVE.L #1200,24(A1)
~040032 JSR -1C8(A6)
~040036 MOVEA.L 00000004,A6
~04003C MOVE.L #CF850,D0
~040042 MOVEQ #4,D1
~040044 JSR -C6(A6)
~040048 TST.L D0
~04004A BEQ 0004005E
~04004E MOVE.L #CF850,D1
~040054 MOVE.W #F0,D2
~040058 JMP 00070000
=====
~04005E MOVE.L #9EB10,D0
~040064 MOVEQ #2,D1
~040066 JSR -C6(A6)
~04006A TST.L D0
~04006C BEQ 00040082
~040070 MOVE.L #80000,D0
~040076 MOVE.L D0,D1

```

Figure 3a – boot block disassembly (1)

```

~040076 MOVE.L D0,D1
~040078 MOVE.W #F00,D2
~04007C JMP 00070000
=====
λ 040082 MOVE.L #61A80,D0
~040088 MOVEQ #4,D1
~04008A JSR -C6(A6)
~04008E TST.L D0
~040090 BEQ 000400A4
~040094 MOVE.L #61A80,D1
~04009A MOVE.W #F,D2
~04009E JMP 00070000
=====
λ 0400A4 MOVEQ #0,D0
~0400A6 MOVEQ #0,D1
~0400A8 MOVEQ #0,D2
~0400AA JMP 00070000
=====
λ 0400B0 ORI.B #0,D0
~0400B4 ORI.B #0,D0
~0400B8 ORI.B #0,D0
~0400BC ORI.B #0,D0
~0400C0 ORI.B #0,D0
~0400C4 ORI.B #0,D0
~0400C8 ORI.B #0,D0

```

Figure 3b – boot block disassembly (2)

First of all we don't see much. Keep going and at \$70070 we will start to see something more interesting. Then we see some basic set up code and a couple of calls to \$700e0 followed by a jmp to \$400. This is interesting as we know we don't currently have anything interesting loaded to \$400 so these calls must either load more data from the disk or copy to \$400 from somewhere else in memory.

Let's take the first call to \$700e0 first. It loads \$60000 into a0 and \$61000 into a1. Since we don't have anything in memory at this point it's a good bet it's going to store something there. We'll now start the original bootblock code executing, put a breakpoint at \$40048 (BS \$40048). We then use G 4000C to start executing the bootblock. When the code stops at \$40048 we next put a breakpoint at \$700c4 and let it run again (use command X) then we can see what's in memory at \$60000 and \$61000. (See figure 5a, 5b, 5c, 5d, 5e and 5f).

This is interesting we see something at \$60000 that looks like a file table and at \$61000 we see something that looks very much like raw mfm data (the many \$aaaa words and \$4489 near the start are a dead giveaway). The mfm data is not of much interest (other than it suggests that the code we have just called is a load routine of some kind).

Looking at the file table in more detail it clearly starts with a list of files of some kind but then there appears to be a second part at \$60c00 that we can't immediately identify. We should now save off the data between \$60000 and \$61000 to a blank disk for later. (See figure 5g).

Now if we return to our disassembly from the current address we see another call to \$700e0 passing in \$70610 and \$400. Now that we have an idea that \$700e0 is a load routine of some kind we can probably assume that \$400 will be the load address. If we get a memory dump of \$70610 (See figure 7) we see what looks like it is probably a filename – which seems to make sense as it's a file listed in the file table we saw at \$60000.

```
trans 40400 41600 70000
Ready.
d 70000
~070000 ORI.B #0,D0
~070004 ORI.B #0,D0
~070008 ORI.B #0,D0
~07000C ORI.B #0,D0
~070010 ORI.B #0,D0
~070014 ORI.B #0,D0
~070018 ORI.B #0,D0
~07001C ORI.B #0,D0
~070020 ORI.B #0,D0
~070024 ORI.B #0,D0
~070028 ORI.B #0,D0
~07002C ORI.B #0,D0
~070030 ORI.B #0,D0
~070034 ORI.B #0,D0
~070038 ORI.B #0,D0
~07003C ORI.B #0,D0
~070040 ORI.B #0,D0
~070044 ORI.B #0,D0
~070048 ORI.B #0,D0
~07004C ORI.B #0,D0
```

Figure 4a – disassembly 2 (1)

```

~07004C ORI.B    #0,D0
~070050 ORI.B    #0,D0
~070054 ORI.B    #0,D0
~070058 ORI.B    #0,D0
~07005C ORI.B    #0,D0
~070060 ORI.B    #0,D0
~070064 ORI.B    #0,D0
~070068 ORI.B    #0,D0
~07006C ORI.B    #0,D0
~070070 MOVEA.L  #80000,A7
~070076 MOVEA.L  #DFF000,A6
~07007C MOVE.W   #7FFF,9A(A6)
~070082 MOVE.W   #7FFF,96(A6)
~070088 MOVE.W   #7FFF,9E(A6)
~07008E MOVE.L   D0,00000004
~070094 MOVE.L   D1,00000008
~07009A MOVEA.L  #400,A0
~0700A0 MOVE.L   #1BF00,D0
~0700A6 CLR.L    (A0)+
~0700A8 SUBQ.L   #1,D0
~0700AA BNE      000700A6
~0700AE LEA      705F0(PC),A5
~0700B2 LEA      00060000,A0
~0700B8 LEA      1000(A0),A1
~0700BC MOVEQ    #0,D0

```

Figure 4b – disassembly 2(2)

```

~0700BC MOVEQ    #0,D0
~0700BE MOVEQ    #5,D1
~0700C0 BSR      000700E0
~0700C4 LEA      70610(PC),A0
~0700C8 MOVEA.L  #400,A1
~0700CE MOVEQ    #0,D1
~0700D0 BSR      000700E0
~0700D4 TST.W    D0
~0700D6 BMI      000700C4
~0700DA JMP      00000400
=====
~0700E0 MOVEM.L  D2-D7/A0-A6,-(A7)
~0700E4 MOVEA.L  #DFF000,A6
~0700EA CLR.W    14(A5)
~0700EE MOVE.W   10(A6),16(A5)
~0700F4 CLR.B    1(A5)
~0700F8 CLR.B    0(A5)
~0700FC CLR.L    C(A5)
~070100 MOVE.W   #FFFF,10(A5)
~070106 SUBQ.W   #1,D1
~070108 BMI      00070160
~07010C SUBQ.W   #1,D1
~07010E BMI      0007015E
~070112 SUBQ.W   #1,D1
~070114 BMI      00070180

```

Figure 4c – disassembly 2 (3)

```

bs 40048
Breakpoint inserted
Ready.
g 4000c
No known virus in memory!
Ready.
Breakpoint raised at address: 00040048
bs 700c4
Breakpoint inserted
Ready.
x
No known virus in memory!
Ready.
Breakpoint raised at address: 000700C4
-

```

Figure 5a – run code until after first call to \$700e0

```

n 60000
:060000 52 65 73 65 72 76 65 64 00 00 00 00 00 00 40 00 Reserved.....@.
:060010 6D 61 69 6E 00 FF FF FF FF FF FF 00 00 70 08 main.....p.
:060020 50 73 79 67 6E 6F 73 69 73 00 FF FF 00 00 4B C8 Psygnosis.....K.
:060030 64 6D 61 6C 6F 67 6F 00 FF FF FF FF 00 00 57 3E dnalogo.....W>
:060040 72 61 6D 70 69 63 00 FF FF FF FF FF 00 00 20 B4 rampic.....
:060050 6E 6F 69 73 65 73 00 FF FF FF FF FF 00 01 9C 06 noises.....
:060060 69 6E 74 72 6F 64 61 74 61 00 FF FF 00 01 D3 FC introdata.....u
:060070 69 6E 74 72 6F 61 6E 69 6D 00 FF FF 00 04 29 D4 introanin.....).
:060080 43 6F 64 65 00 FF FF FF FF FF FF 00 01 06 C4 Code.....A
:060090 65 6E 64 70 69 63 00 FF FF FF FF FF 00 00 73 96 endpic.....s.
:0600A0 65 6E 64 73 61 6D 70 6C 65 00 FF FF 00 01 25 EA endsample.....%.
:0600B0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF .....
:0600C0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF .....
:0600D0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF .....
:0600E0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF .....
:0600F0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF .....
:060100 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF .....
:060110 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF .....
:060120 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF .....
:060130 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF .....
:060140 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF .....
:060150 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF .....
:060160 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF .....

```

Figure 5b – memory dump starting at \$60000

```

:060BD0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF .....
:060BE0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF .....
:060BF0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF .....
:060C00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
:060C10 01 01 01 01 01 01 01 01 01 01 01 01 01 01 .....
:060C20 01 01 01 01 01 01 01 01 01 01 01 01 02 02 .....
:060C30 02 02 02 02 02 02 02 02 02 02 02 02 02 02 .....
:060C40 03 03 03 03 03 03 03 03 03 03 03 03 03 03 .....
:060C50 03 03 03 03 03 03 04 04 04 04 04 04 04 05 .....
:060C60 05 05 05 05 05 05 05 05 05 05 05 05 05 05 .....
:060C70 05 05 05 05 05 05 05 05 05 05 05 05 05 05 .....
:060C80 05 05 05 05 05 05 05 05 05 05 05 05 05 05 .....
:060C90 05 05 05 05 05 05 05 05 05 05 05 05 05 05 .....
:060CA0 05 05 05 05 05 05 05 05 05 05 05 05 05 05 .....
:060CB0 05 05 05 05 05 05 05 05 05 05 05 05 05 05 .....
:060CC0 05 05 05 05 05 05 05 06 06 06 06 06 06 06 .....
:060CD0 06 06 06 06 06 06 06 06 06 06 06 06 06 06 .....
:060CE0 06 06 06 06 06 06 06 06 06 06 06 06 06 06 .....
:060CF0 06 06 06 06 06 06 06 06 06 06 06 06 06 06 .....
:060D00 06 06 06 06 06 06 06 06 06 06 06 06 06 06 .....
:060D10 06 06 06 06 06 06 06 06 06 06 06 06 06 06 .....
:060D20 06 06 06 06 06 06 06 06 06 06 06 06 06 06 .....
:060D30 06 06 06 06 06 06 06 06 06 06 06 07 07 07 .....
:060D40 07 07 07 07 07 07 07 07 07 07 07 07 07 07 .....
:060D50 07 07 07 07 07 07 07 07 07 07 07 07 07 07 .....

```

Figure 5c – continued memory dump (1)

```

:060D60 07 07 07 07 07 07 07 07 07 07 07 07 07 07 .....
:060D70 07 07 07 07 07 07 07 07 07 07 07 07 07 07 .....
:060D80 07 07 07 07 07 07 07 07 07 07 07 07 07 07 .....
:060D90 07 07 07 07 07 07 07 07 07 07 07 07 07 07 .....
:060DA0 07 07 07 07 07 07 07 07 07 07 07 07 07 07 .....
:060DB0 07 07 07 07 07 07 07 07 07 07 07 07 07 07 .....
:060DC0 07 07 07 07 07 07 07 07 07 07 07 07 07 07 .....
:060DD0 07 07 07 07 07 07 07 07 07 07 07 07 07 07 .....
:060DE0 07 07 07 07 07 07 07 07 07 07 07 07 07 07 .....
:060DF0 07 07 07 07 07 07 07 07 07 07 07 07 07 07 .....
:060E00 07 07 07 07 07 07 07 07 07 07 07 07 07 07 .....
:060E10 07 07 07 07 07 07 07 07 07 07 07 07 07 07 .....
:060E20 07 07 07 07 07 07 07 07 07 07 07 07 07 07 .....
:060E30 07 07 07 07 07 07 07 07 07 07 07 07 07 07 .....
:060E40 07 07 07 07 07 07 07 08 08 08 08 08 08 08 .....
:060E50 08 08 08 08 08 08 08 08 08 08 08 08 08 08 .....
:060E60 08 08 08 08 08 08 08 08 08 08 08 08 08 08 .....
:060E70 08 08 08 08 08 08 08 08 08 08 08 08 08 08 .....
:060E80 08 08 08 08 08 08 08 08 09 09 09 09 09 09 .....
:060E90 09 09 09 09 09 09 09 09 09 09 09 09 09 09 .....
:060EA0 09 09 09 09 09 09 0A 0A 0A 0A 0A 0A 0A 0A .....
:060EB0 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A .....
:060EC0 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A .....
:060ED0 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A .....
:060EE0 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A .....

```

Figure 5d – continued memory dump (2)

```

:060EE0 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A .....
:060EF0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF .....
:060F00 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF .....
:060F10 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF .....
:060F20 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF .....
:060F30 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF .....
:060F40 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF .....
:060F50 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF .....
:060F60 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF .....
:060F70 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF .....
:060F80 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF .....
:060F90 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF .....
:060FA0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF .....
:060FB0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF .....
:060FC0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF .....
:060FD0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF .....
:060FE0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF .....
:060FF0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF .....
:061000 AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA .....
:061010 AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA .....
:061020 AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA .....
:061030 AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA .....
:061040 AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA .....
:061050 AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA .....
:061060 AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA .....

```

Figure 5e – continued memory dump (3)

```

:061060 AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA .....
:061070 AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA .....
:061080 AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA .....
:061090 AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA .....
:0610A0 44 89 55 2A AA AA 49 2A A9 44 A9 12 52 45 11 12 D,U*,I*,D.,RE..
:0610B0 51 45 11 11 52 54 92 92 45 44 AA AA AA AA AA AA OE.,RT.,ED.....
:0610C0 AA AA AA AA AA AA AA AA 4A AA 94 92 45 49 14 95 .....J.,EI..
:0610D0 49 44 AA 55 2A 55 55 55 55 55 55 55 55 55 55 ID,U*UUUUUUUUUU
:0610E0 55 55 2A AA AA AA 92 A4 52 AA 91 52 51 14 91 UU*,...,R.,..RQ..
:0610F0 51 45 15 15 44 45 11 14 51 49 11 2A 51 2A 55 55 OE.,DE.,OI,*OU
:061100 55 55 2A AA AA AA A5 44 49 4A 92 94 44 45 12 94 UU*,...,DIJ.,DE..
:061110 49 44 95 11 45 45 15 2A 45 2A 55 55 55 55 55 55 ID.,EE.*E*UUUUUU
:061120 55 55 2A AA AA AA A9 15 55 14 91 12 52 49 14 92 UU*,...,U.,..RI..
:061130 45 52 94 91 49 49 2A 55 2A 55 55 55 55 55 55 ER.,II*UUUUUUUU
:061140 55 55 2A AA AA AA 92 52 AA 94 95 15 44 45 14 91 UU*,...,R.,..DE..
:061150 49 51 12 91 45 51 2A 55 2A 55 55 55 55 55 55 IO.,EQ*U*UUUUUUUU
:061160 55 55 2A AA AA AA A9 44 A9 14 A4 94 95 49 44 92 91 UU*,...,D.,...,ID..
:061170 54 52 95 12 45 44 92 92 49 54 92 AA 49 2A 55 55 TR.,ED.,IT.,I*UU
:061180 55 55 2A AA AA A9 49 54 51 54 94 95 49 44 92 91 UU*,...,ITOT.,ID..
:061190 54 52 95 12 45 49 15 14 44 49 14 AA 45 2A 55 55 TR.,EI.,DI.,E*UU
:0611A0 55 55 2A AA AA AA 94 94 A9 54 A9 15 49 45 12 92 UU*,...,J.T.,IE..
:0611B0 44 45 2A 55 2A 55 55 55 55 55 55 55 55 55 55 DE*U*UUUUUUUUUUUU
:0611C0 55 55 2A AA AA A9 29 4A A4 44 92 95 45 44 92 92 UU*,...)J.D.,ED..
:0611D0 44 52 94 91 49 49 2A 55 2A 55 55 55 55 55 55 DR.,II*U*UUUUUUUU
:0611E0 55 55 2A AA AA AA 91 49 51 14 92 95 45 44 92 91 UU*,...,IQ.,..ED..

```

Figure 5f – continued memory dump (4)

```

:061180 55 55 2A AA AA A9 49 54 51 54 74 95 49 44 92 91 00*...I...ED..
:061190 54 52 95 12 45 49 15 14 44 49 14 AA 45 2A 55 55 TR..EI..DI..E*UU
:0611A0 55 55 2A AA AA A4 94 4A A9 54 A9 15 49 45 12 92 UU*...J.T..IE..
:0611B0 44 45 2A 55 2A 55 55 55 55 55 55 55 55 55 55 DE*U*UUUUUUUUUU
:0611C0 55 55 2A AA AA A9 29 4A A4 44 92 95 45 44 92 92 UU*...J.D..ED..
:0611D0 44 52 94 91 49 49 2A 55 2A 55 55 55 55 55 55 DR..II*U*UUUUUU
:0611E0 55 55 2A AA AA AA 91 49 51 14 92 95 45 44 92 91 UU*...I...ED..

sn lemmings_d1_filetable, 60000 61000
Disk ok
-

```

Figure 5g – save file table

So we can see that the same routine is being called twice and it seems to have different results each time. Looking at the routine starting at \$700e0 we can see that very soon after it is called it branches off to different parts of the code based on the value in D1. (Starting at \$70106). We can see that from the calls we are seeing so far that 5 in D1 will load up the file table and 0 in D1 will load a file. What the other values in D1 may do at this stage we don't know. They don't seem to be used in this part of the code anyhow so at this stage we don't need to worry about this right now.

Now that we have a very basic understanding of this load routine we can put a breakpoint at \$700D4 (BS \$700D4) and we will see if our ideas so far are correct. Let the code continue running (use command X). Now we can get a disassembly of the memory at \$400 and see what we can see. (See figure 6a). Definitely looks like we have something loaded right here.

```

bs 700d4
Breakpoint inserted
Ready.
X
No known virus in memory!
Ready.
Breakpoint raised at address: 000700D4
d 400
~000400 MOVEA.L #DFF000,A6
~000406 CLR.W 180(A6)
~00040A MOVE.W #7FFF,9A(A6)
~000410 MOVE.W #7FFF,96(A6)
~000416 MOVE.L #FFFFFFFF,44(A6)
~00041E MOVE.B #7F,00BFED01
~000426 MOVE.B #7F,00BFDD00
~00042E MOVE.B 00BFED01,D0
~000434 LEA 44E(PC),A0
~000438 LEA 00078AD0,A1
~00043E MOVE.L A1,00000084.S
~000442 MOVE.W #37DC,D0
~000446 MOVE.W (A0)+,(A1)+
~000448 DBF D0,00000446
~00044C TRAP #1
~00044E LEA 111E(PC),A5

```

Figure 6a – disassembly of main file loaded at address \$400

```

n 73f0
:0073f0 67 F6 52 43 B6 40 66 EA 4C DF 00 0E 4E 75 6E 6F göRC.0f.Lß..Nuno
:007400 69 73 65 73 00 00 00 44 00 00 00 00 00 00 00 00 ises...D.....
:007410 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
:007420 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
:007430 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
:007440 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
:007450 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
:007460 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
:007470 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
:007480 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
:007490 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
:0074A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
:0074B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
:0074C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....

```

Extracting the MFM data

Now that we have found a load routine what we want to try and do is see if we can use this load routine to try and extract all of the disk data (or even better both disks). If we are going to save this data to an AmigaDOS disk in files we will need 3 blank disks in order to fit the data onto.

Figure 6b – memory dump from the end of the loaded file

What we need to figure out is how we can fool this load routine to loading the data we want to extract. We want to try and extract the first disk in two chunks. We will be looking to extract tracks 0 – 39 and then 60 – 79 and save the two areas of memory onto two of our blank disks.

Taking a closer look at the load table in figures 5a – 5e we see at the start a list of files but then there is only four other bytes of additional information for each file eg \$7008 in the case of the **main** file. We would normally expect a disk location (eg. Start sector or similar) and a file length. So we don't quite know how this load routine calculates these two things from this file entry. Since we currently have the main file loaded we can have a quick look and see where this file ends and see if that gives us a clue. If \$7008 is the length of the file then if we take a look at \$7008+\$400 we should see the end of the file. (see Figure 6b). I dump from \$73F0 and it definitely looks like we are onto something here.

So if the values listed in the file table are the file lengths we can quickly total them up as a double check to see if we end up with something sensible looking. A quick calculation gives around 747k so that seems reasonable to me. So what we don't know now is how the loader calculates the correct position on the disk to load the file from. Looking at the load routine in more detail we see that the main code for the file loader starts at \$70160 (See figure 7a).

```

~070160 BSR      0007054A
~070164 BSR      00070292
~070168 TST.W    D1
~07016A BEQ      00070178
~07016E BSR      000702D6
~070172 MOVE.L   C(A5),D1
~070176 BRA      00070142
;=====
^070178 MOVE.W   #FFFF,14(A5)
~07017E BRA      00070142
;=====
^070180 BSR      0007054A
~070184 BSR      0007018A
~070188 BRA      00070142
;=====
^07018A MOVEM.L  D0-D1/A1,-(A7)
~07018E BSR      000704F8
~070192 BSET     #2,00BFD100
~07019A CLR.W    10(A5)
~07019E BSR      00070424
~0701A2 TST.W    14(A5)
~0701A6 BNE      000701EE
~0701AA MOVEQ    #0,D1
~0701AC MOVEA.L  4(A5),A1
~0701B0 MOVE.W   #400,D7

```

Figure 7a – load routine(1)

```

d 70292
~070292 MOVEM.L  D2/D6-D7/A0-A1,-(A7)
~070296 MOVEA.L  4(A5),A1
~07029A MOVE.W   #BF,D7
~07029E MOVEQ    #0,D1
~0702A0 MOVEQ    #B,D6
~0702A2 MOVEM.L  A0-A1,-(A7)
~0702A6 MOVE.B   (A0)+,D2
~0702A8 CMP.B    (A1)+,D2
~0702AA BNE      000702C0
~0702AE TST.B    D2
~0702B0 BEQ      000702B8
~0702B4 DBF      D6,000702A6
~0702B8 MOVEM.L  (A7)+,A0-A1
~0702BC BRA      000702D0
;=====
^0702C0 ADDQ.W   #1,D1
~0702C2 MOVEM.L  (A7)+,A0-A1
~0702C6 LEA      10(A1),A1
~0702CA DBF      D7,000702A0
~0702CE MOVEQ    #0,D1
~0702D0 MOVEM.L  (A7)+,D2/D6-D7/A0-A1
~0702D4 RTS

```

Figure 7b – load routine continued(2)

```

~0702D6 MOVEA.L 4(A5),A0
~0702DA LSL.W #4,D1
~0702DC ADDA.W D1,A0
~0702DE LSR.W #4,D1
~0702E0 MOVE.L C(A0),D2
~0702E4 MOVE.L D2,C(A5)
~0702E8 MOVEA.L 4(A5),A0
~0702EC LEA C00(A0),A0
~0702F0 MOVEQ #0,D3
~0702F2 CMP.B (A0)+,D1
~0702F4 BNE 00070388
~0702F8 MOVE.L D3,D6
~0702FA DIVU.W #C,D6
~0702FE SWAP D6
~070300 MOVE.W D6,D4
~070302 SWAP D6
~070304 MOVEQ #0,D5
~070306 CMP.W #5,D4
~07030A BLE 00070312
~07030E MOVEQ #1,D5
~070310 SUBQ.W #6,D4
~070312 CMP.W 12(A5),D6
~070316 BNE 00070326
~07031A CMP.W 10(A5),D5
~07031E BNE 00070326

```

Figure 7c – load routine continued(3)

```

~07031E BNE 00070326
~070322 BRA 00070358
=====
~070326 MOVE.W D6,D0
~070328 BSET #2,00BFD100
~070330 CLR.W 10(A5)
~070334 TST.W D5
~070336 BEQ 00070348
~07033A BCLR #2,00BFD100
~070342 MOVE.W #1,10(A5)
~070348 BSR 00070496
~07034C BSR 00070424
~070350 TST.W 14(A5)
~070354 BNE 0007038E
~070358 MOVE.L D1,-(A7)
~07035A MOVE.W D4,D1
~07035C MOVE.W #400,D7
~070360 CMP.L #400,D2
~070366 BGE 0007036C
~07036A MOVE.W D2,D7
~07036C BSR 00070390
~070370 MOVE.L (A7)+,D1
~070372 TST.W 14(A5)
~070376 BNE 0007038E
~07037A LEA 400(A1),A1

```

Figure 7d – load routine continued (4)

```

~07037E SUBI.L #400,D2
~070384 BLE 0007038E
~070388 ADDQ.W #1,D3
~07038A BRA 000702F2
=====
~07038E RTS
=====

```

Figure 7e – load routine continued (5)

Looking at the routines called we see nothing much of interest at \$7054a (just some drive select stuff). Looking at \$70292 (See figure 7b) we see something a little more interesting. Some code that looks like its comparing something (a filename?) and then skipping \$10 bytes if it doesn't match. This looks to me like a routine to find our file in the file table given the filename. It returns the file entry in D1 or 0 if it can't be found. Now we look at the next routine (Figure 7c, 7d, 7e). This is interesting again as the first thing its doing is using our file number, doing an LSL #4 which multiplies by \$10 and then pulls out the value at that offset plus \$C into D2 so that's pulling out the length and storing it. Next we see that we pull out what appears to be the start of the file table and adds \$C00 so this points us to the second part of the file table that we currently don't understand.

It then searches through that part of the file table looking for the file number that is stored in D1.

Now that we know that the numbers in the second part of the file table correspond to the files in the first part we can theorise that it might be a file bitmap. eg. Each byte identifies which file it belongs to. So by finding the first byte with the matching file number it is locating the start of the file. If we want to extract the disk data we can do it by manipulating this file table to fool the loader into pulling the whole disk into ram. If we look at the file table again and for the **main** file we see that the file size is \$7008 and the file number 1 occupies 29 blocks in the second part of the file table. If we divide \$7008 by 29 we get \$3DC so we can probably assume that each block in the file table is \$400 (1024) bytes. This is confirmed by the **reserved** file that is \$4000 bytes and occupies the first 16 blocks ($16 * \$400 = \4000). If we calculate the total number of blocks (starting at \$60c10 and ending at \$60ef0 that gives us a total of \$2e0 blocks. We will try to load this data in two halves. \$2e0 blocks / 2 is \$170 blocks or \$5c000 bytes.

If we manipulate the file table so that the **main** file is \$5C000 bytes long and occupies \$170 of the blocks we should end up loading \$5C000 bytes from the disk. If we load this at address \$400 we should have enough room in memory to store this data.

See figure8a and 8b for the command to fill the memory in the file table and the results. Then we modify the file length in the first part of the file table. (see Figure 8c)

```
o 1, 60c10 60d80
Ready.
n 60c00
:060C00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
:060C10 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 .....
:060C20 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 .....
:060C30 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 .....
:060C40 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 .....
:060C50 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 .....
:060C60 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 .....
:060C70 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 .....
:060C80 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 .....
:060C90 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 .....
:060CA0 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 .....
:060CB0 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 .....
:060CC0 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 .....
:060CD0 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 .....
:060CE0 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 .....
:060CF0 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 .....
:060D00 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 .....
:060D10 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 .....
:060D20 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 .....
:060D30 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 .....
:060D40 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 .....
```

Figure 8a – file table – modified (1)

```
:060D50 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 .....
:060D60 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 .....
:060D70 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 .....
:060D80 07 07 07 07 07 07 07 07 07 07 07 07 07 07 07 07 .....
:060D90 07 07 07 07 07 07 07 07 07 07 07 07 07 07 07 07 .....
:060DA0 07 07 07 07 07 07 07 07 07 07 07 07 07 07 07 07 .....
:060DB0 07 07 07 07 07 07 07 07 07 07 07 07 07 07 07 07 .....
:060DC0 07 07 07 07 07 07 07 07 07 07 07 07 07 07 07 07 .....
:060DD0 07 07 07 07 07 07 07 07 07 07 07 07 07 07 07 07 .....
:060DE0 07 07 07 07 07 07 07 07 07 07 07 07 07 07 07 07 .....
```

Figure 8b – file table - modified (2)

```
n 60000
:060000 52 65 73 65 72 76 65 64 00 00 00 00 00 00 40 00 Reserved.....0.
:060010 6D 61 69 6E 00 FF FF FF FF FF FF 00 05 C0 00 main.....
:060020 50 73 79 67 6E 6F 73 69 73 00 FF FF 00 00 4B C8 Psygnosis.....K.
:060030 64 6D 61 6C 6F 67 6F 00 FF FF FF FF 00 00 57 3E dmlogo.....W>
:060040 72 61 6D 70 69 63 00 FF FF FF FF FF 00 00 20 B4 rampic.....
:060050 6E 6F 69 73 65 73 00 FF FF FF FF FF 00 01 9C 06 noises.....
:060060 69 6E 74 72 6F 64 61 74 61 00 FF FF 00 01 D3 FC introdada.....ü
:060070 69 6E 74 72 6F 61 6E 69 6D 00 FF FF 00 04 29 D4 introanim.....).
:060080 43 6F 64 65 00 FF FF FF FF FF FF 00 01 06 C4 Code.....Ä
:060090 65 6E 64 70 69 63 00 FF FF FF FF FF 00 00 73 96 endpic.....s.
```

Figure 8c – file table – file size modified

```
r pc 700c4
D0-00000000 00007008 000000F0 00000000 00000000 00000F4F FFFFFFFF 0000FFFF
A0-00070610 00000400 00000001 00202EB0 00002E20 000705F0 00DFF000 00080000
PC = 000700C4 USP = 00080000 SR = 0004 T=0 S=0 I=000 X=0 N=0 Z=1 V=0 C=0
-
```

Figure 8d – re run loader

```
sm lemnings_d1_p1, 400 5c400
Disk ok
-
```

Figure 8e – save disk 1 part 1

```

o 2,60c10 60d80
Ready.

o 1, 60d80 60ef0
Ready.

m 60d70
:060D70 02 02 02 02 02 02 02 02 02 02 02 02 02 02 02 02 .....
:060D80 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 .....
:060D90 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 .....
:060DA0 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 .....
:060DB0 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 .....
:060DC0 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 .....
:060DD0 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 .....
:060DE0 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 .....
:060DF0 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 .....
:060E00 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 .....
:060E10 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 .....
:060E20 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 .....
:060E30 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 .....
:060E40 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 .....
:060E50 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 .....
:060E60 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 .....
:060E70 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 .....

```

Figure 8f – file table modified to load second half of disk 1 (1)

```

:060E70 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 .....
:060E80 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 .....
:060E90 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 .....
:060EA0 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 .....
:060EB0 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 .....
:060EC0 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 .....
:060ED0 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 .....
:060EE0 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 .....
:060EF0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF .....
:060F00 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF .....
:060F10 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF .....
:060F20 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF .....

```

Figure 8g– file table modified to load second half of disk 1 (2)

Now we are ready to rerun the load routine with the relevant values. Reset the PC to \$700c4 (see Figure 8d) then you can let the code run again to the breakpoint at \$700D4. Once the breakpoint hits you can save the memory from \$400 to \$5C400 (see figure 8e) to our first blank floppy. Now we can update the file table to extract the second half of the data. (See figure 8f, 8g) and re-run the code (don't forget to re-insert the lemmings disk 1 first) and save out the second half of the data to the blank floppy (see figure 8h).

```

r pc 700c4
D0=FFFFFFFF 0005C000 000000F0 00000000 00000000 00000F4F FFFFFFFF 0000FFFF
A0=00070610 00000400 00000001 00202EB0 00002E20 000705F0 00DFF000 00080000
PC = 000700C4 USP = 00080000 SR = 0008 T=0 S=0 I=000 X=0 N=1 Z=0 V=0 C=0
x
No known virus in memory!
Ready.
Breakpoint raised at address: 000700D4
r
D0=00000000 0005C000 000000F0 00000000 00000000 00000F4F FFFFFFFF 0000FFFF
A0=00070610 00000400 00000001 00202EB0 00002E20 000705F0 00DFF000 00080000
PC = 000700D4 USP = 00080000 SR = 0004 T=0 S=0 I=000 X=0 N=0 Z=1 V=0 C=0

sm lemmings_d1_p2, 400 5c400
Disk ok

```

Figure 8h

Now we want to try and get the second disk in the same way as the first disk. We will need to restart the code from the point that loads in the file table so we can see what's on disk 2. Insert disk 2 and reset the code to \$700AE and put a breakpoint at \$700C4 so we can look at the file table. (see figure 9a) then let the code execute to the breakpoint. Now dump the file table memory (see figure 9a to 9j). Save the file table off to a blank disk for use later (see figure 9k)

```

r pc 700ae
D0=00000000 0005C000 000000F0 00000000 00000000 00000F4F FFFFFFFF 0000FFFF
A0=00070610 00000400 00000001 00202EB0 00002E20 000705F0 00DFF000 00080000
PC = 000700AE USP = 00080000 SR = 0004 T=0 S=0 I=000 X=0 N=0 Z=1 V=0 C=0
bs 700c4
Breakpoint inserted
Ready.
x
No known virus in memory!
Ready.
Breakpoint raised at address: 000700C4
-

```

Figure 9a - Read file table from disk 2

```

n 60000
:060000 52 65 73 65 72 76 65 64 00 00 00 00 00 00 40 00 Reserved.....0.
:060010 47 72 6F 75 6E 64 31 00 FF FF FF FF 00 00 38 20 Ground1.....8
:060020 4F 62 6A 65 63 74 73 31 00 FF FF FF FF 00 00 28 A8 Objects1.....(
:060030 47 72 6F 75 6E 64 32 00 FF FF FF FF 00 00 4A EC Ground2.....J
:060040 4F 62 6A 65 63 74 73 32 00 FF FF FF FF 00 00 36 5C Objects2.....6\
:060050 47 72 6F 75 6E 64 33 00 FF FF FF FF 00 00 33 20 Ground3.....3
:060060 4F 62 6A 65 63 74 73 33 00 FF FF FF FF 00 00 24 50 Objects3.....$P
:060070 47 72 6F 75 6E 64 34 00 FF FF FF FF 00 00 30 58 Ground4.....0X
:060080 4F 62 6A 65 63 74 73 34 00 FF FF FF FF 00 00 29 4C Objects4.....)L
:060090 47 72 6F 75 6E 64 35 00 FF FF FF FF 00 00 45 F0 Ground5.....E
:0600A0 4F 62 6A 65 63 74 73 35 00 FF FF FF FF 00 00 2A E8 Objects5.....*.
:0600B0 6C 65 76 65 6C 64 61 74 61 00 FF FF 00 00 12 94 leveldata.....
:0600C0 49 63 6F 6E 73 00 FF FF FF FF FF 00 01 35 8A Icons.....5.
:0600D0 6F 64 64 73 71 62 6C 65 00 FF FF FF 00 00 11 80 oddtable.....
:0600E0 70 61 6E 65 6C 31 00 FF FF FF FF FF 00 00 0D 14 panel1.....
:0600F0 70 61 6E 65 6C 32 00 FF FF FF FF FF 00 00 0C 78 panel2.....x
:060100 61 77 65 73 6F 6D 65 00 FF FF FF FF 00 00 0A 64 awesome.....d
:060110 62 65 61 73 74 49 00 FF FF FF FF 00 00 0B 30 beastI.....0
:060120 62 65 61 73 74 49 00 FF FF FF FF 00 00 12 98 beastII.....
:060130 63 61 6E 63 61 6E 00 FF FF FF FF 00 00 09 3C cancan.....<
:060140 64 6F 67 67 69 65 00 FF FF FF FF 00 00 07 90 doggie.....
:060150 6D 65 6E 61 63 65 00 FF FF FF FF 00 00 18 AC menace.....
:060160 6D 6F 75 6E 74 61 69 6E 00 FF FF FF 00 00 06 98 mountain.....
:060170 74 65 6E 6C 65 6D 6D 69 6E 67 73 00 00 00 06 70 tenlemnings....p

```

Figure 9b – Memory dump of file table from disk 2(1)

```

:060180 74 69 6D 31 00 FF FF FF FF FF 00 00 05 88 tim1.....
:060190 74 69 6D 32 00 FF FF FF FF FF 00 00 05 38 tim2.....8
:0601A0 74 69 6D 33 00 FF FF FF FF FF 00 00 0C 54 tim3.....T
:0601B0 74 69 6D 34 00 FF FF FF FF FF 00 00 13 14 tim4.....
:0601C0 74 69 6D 35 00 FF FF FF FF FF 00 00 14 0C tim5.....
:0601D0 74 69 6D 36 00 FF FF FF FF FF 00 00 0D 50 tim6.....P
:0601E0 74 69 6D 37 00 FF FF FF FF FF 00 00 0D B4 tim7.....
:0601F0 74 69 6D 38 00 FF FF FF FF FF 00 00 0E 50 tim8.....P
:060200 74 69 6D 39 00 FF FF FF FF FF 00 00 09 A8 tim9.....
:060210 74 69 6D 31 30 00 FF FF FF FF FF 00 00 11 0C tim10.....
:060220 6C 65 6D 6D 69 6E 67 31 00 FF FF FF 00 00 0D A0 lemming1.....
:060230 6C 65 6D 6D 69 6E 67 32 00 FF FF FF 00 00 1A 04 lemming2.....
:060240 6C 65 6D 6D 69 6E 67 33 00 FF FF FF 00 00 0F 20 lemming3.....
:060250 4C 65 76 65 6C 30 30 30 00 FF FF FF 00 00 05 34 Level000.....4
:060260 4C 65 76 65 6C 30 30 31 00 FF FF FF 00 00 09 E0 Level001.....
:060270 4C 65 76 65 6C 30 30 32 00 FF FF FF 00 00 04 A0 Level002.....
:060280 4C 65 76 65 6C 30 30 33 00 FF FF FF 00 00 0D 94 Level003.....
:060290 4C 65 76 65 6C 30 30 34 00 FF FF FF 00 00 0A 88 Level004.....
:0602A0 4C 65 76 65 6C 30 30 35 00 FF FF FF 00 00 08 50 Level005.....P
:0602B0 4C 65 76 65 6C 30 30 36 00 FF FF FF 00 00 0E CC Level006.....
:0602C0 4C 65 76 65 6C 30 30 37 00 FF FF FF 00 00 0D 48 Level007.....H
:0602D0 4C 65 76 65 6C 30 30 38 00 FF FF FF 00 00 07 CC Level008.....
:0602E0 4C 65 76 65 6C 30 30 39 00 FF FF FF 00 00 07 6C Level009.....l
:0602F0 4C 65 76 65 6C 30 31 30 00 FF FF FF 00 00 0F 48 Level010.....H
:060300 4C 65 76 65 6C 30 31 31 00 FF FF FF 00 00 0D DC Level011.....ü

```

Figure 9c – Memory dump of file table from disk 2(2)

```

:060310 4C 65 76 65 6C 30 31 32 00 FF FF FF 00 00 10 14 Level012.....
:060320 4C 65 76 65 6C 30 31 33 00 FF FF FF 00 00 0C C4 Level013.....A
:060330 4C 65 76 65 6C 30 31 34 00 FF FF FF 00 00 09 EC Level014.....
:060340 4C 65 76 65 6C 30 31 35 00 FF FF FF 00 00 0B 10 Level015.....
:060350 4C 65 76 65 6C 30 31 36 00 FF FF FF 00 00 0B AC Level016.....
:060360 4C 65 76 65 6C 30 31 37 00 FF FF FF 00 00 10 98 Level017.....
:060370 4C 65 76 65 6C 30 31 38 00 FF FF FF 00 00 0B 20 Level018.....
:060380 4C 65 76 65 6C 30 31 39 00 FF FF FF 00 00 0B 38 Level019.....8
:060390 4C 65 76 65 6C 30 32 30 00 FF FF FF 00 00 0B E8 Level020.....
:0603A0 4C 65 76 65 6C 30 32 31 00 FF FF FF 00 00 05 8C Level021.....
:0603B0 4C 65 76 65 6C 30 32 32 00 FF FF FF 00 00 09 28 Level022.....(
:0603C0 4C 65 76 65 6C 30 32 33 00 FF FF FF 00 00 14 F8 Level023.....
:0603D0 4C 65 76 65 6C 30 32 34 00 FF FF FF 00 00 07 E0 Level024.....
:0603E0 73 70 65 63 69 61 6C 30 00 FF FF FF 00 00 75 64 special0.....ud
:0603F0 73 70 65 63 69 61 6C 31 00 FF FF FF 00 00 4A 24 special1.....J$
:060400 73 70 65 63 69 61 6C 32 00 FF FF FF 00 00 59 CC special2.....Y.
:060410 73 70 65 63 69 61 6C 33 00 FF FF FF 00 00 87 88 special3.....
:060420 61 63 6F 75 73 74 69 63 00 FF FF FF 00 00 13 1C acoustic.....
:060430 62 6C 6F 63 6B 73 00 FF FF FF 00 00 07 88 blocks.....
:060440 62 72 61 73 73 31 00 FF FF FF 00 00 22 4C brass1....."L
:060450 63 68 65 6C 6C 6F 00 FF FF FF 00 00 0F 60 chello.....
:060460 63 6C 61 70 00 FF FF FF 00 00 05 E0 clap.....
:060470 64 6F 67 62 61 72 6B 31 00 FF FF FF 00 00 0A 5C dogbark1.....\
:060480 64 72 75 6D 00 FF FF FF 00 00 0A 84 drum.....
:060490 65 6C 65 63 62 61 73 73 00 FF FF FF 00 00 0C 6A elecbass.....J

```

Figure 9d– Memory dump of file table from disk 2(3)

```

:060490 65 6C 65 63 62 61 73 73 00 FF FF FF 00 00 0C 6A elecbass.....j
:0604A0 65 6C 65 63 67 75 69 74 61 72 00 FF 00 00 0F E8 elecguitar.....
:0604B0 67 72 75 6E 74 00 FF FF FF 00 00 0F C4 grunt.....A
:0604C0 6A 61 7A 7A 6F 72 67 61 6E 00 FF FF 00 00 07 9C jazzorgan.....
:0604D0 70 69 61 6E 6F 00 FF FF FF 00 00 14 F4 piano.....
:0604E0 70 69 61 6E 6F 31 00 FF FF FF 00 00 0E 42 pianol.....B
:0604F0 73 61 78 00 FF FF FF 00 00 1B 18 sax.....
:060500 73 6C 75 72 64 72 75 6D 00 FF FF 00 00 12 5C slurdun.....\
:060510 73 6E 61 72 65 00 FF FF FF 00 00 0E CE snare.....
:060520 73 70 72 69 6E 67 79 00 FF FF FF 00 00 1E 92 springy.....
:060530 73 74 72 69 6E 67 73 32 00 FF FF FF 00 00 12 BA strings2.....
:060540 73 79 6E 74 68 68 61 72 70 00 FF FF 00 00 12 BC synthharp.....
:060550 73 79 6E 74 68 6C 65 61 64 00 FF FF 00 00 13 58 synthlead.....X
:060560 74 61 63 63 62 61 73 73 00 FF FF 00 00 0D 8C taccbass.....
:060570 74 62 61 73 73 64 72 75 6D 31 00 FF 00 00 0F F6 tbassdrum1.....ö
:060580 74 62 61 73 73 64 72 75 6D 32 00 FF 00 00 0E 3E tbassdrum2.....>
:060590 74 62 65 6C 6C 00 FF FF FF 00 00 17 FC tbell.....ü
:0605A0 74 62 69 67 63 68 6F 72 64 00 FF FF 00 00 26 7C tbigchord.....&l
:0605B0 74 62 75 7A 7A 00 FF FF FF 00 00 21 18 tbuzz.....!
:0605C0 74 63 72 69 6E 68 6C 65 00 FF FF 00 00 18 50 tcrinkle.....P
:0605D0 74 66 62 61 73 73 00 FF FF FF 00 00 15 08 tfbass.....
:0605E0 74 68 61 72 70 00 FF FF FF 00 00 12 84 tharp.....
:0605F0 74 68 75 6D 70 00 FF FF FF 00 00 09 E4 thump.....ä
:060600 74 6B 6D 69 78 00 FF FF FF 00 00 1B F8 tkmix.....
:060610 74 6F 64 64 00 FF FF FF 00 00 1A B4 todd.....

```

Figure 9e– Memory dump of file table from disk 2(4)

```

:060620 74 70 69 61 6E 6F 68 69 00 FF FF FF 00 00 11 2C tpianohi.....
:060630 74 70 69 61 6E 6F 6C 6F 00 FF FF FF 00 00 13 AC tpianolo.....
:060640 74 70 69 7A 7A 00 FF FF FF FF FF 00 00 07 44 tpizz.....D
:060650 74 73 68 61 6B 65 72 00 FF FF FF FF 00 00 06 4A tshaker.....J
:060660 74 73 6E 61 72 65 31 00 FF FF FF FF 00 00 15 82 tsnare1.....
:060670 74 73 6E 61 72 65 32 00 FF FF FF FF 00 00 12 B4 tsnare2.....
:060680 74 73 70 61 63 65 62 61 73 73 00 FF 00 00 10 30 tspacebass....0
:060690 74 73 70 72 69 6E 6B 6C 65 00 FF FF 00 00 2A D4 tsprinkle.....*
:0606A0 74 73 71 72 62 65 6C 6C 00 FF FF FF 00 00 0A DC tsqrbell.....ü
:0606B0 74 73 74 72 69 6E 67 00 FF FF FF FF 00 00 17 42 tstring.....B
:0606C0 74 74 69 6E 6B 00 FF FF FF FF FF 00 00 14 1C ttink.....
:0606D0 74 74 72 75 6D 70 65 74 00 FF FF FF 00 00 13 E4 ttrumpet.....ä
:0606E0 7A 61 70 00 FF FF FF FF FF 00 00 0F 0C zap.....
:0606F0 42 61 73 69 63 46 58 00 FF FF FF FF 00 00 91 40 BasicFX.....@
:060700 46 75 6C 6C 46 58 00 FF FF FF FF FF 00 00 9A 3C FullFX.....<
:060710 61 62 61 73 73 00 FF FF FF FF FF 00 00 09 BC abass.....
:060720 61 62 64 72 00 FF FF FF FF FF 00 00 0A 8A abdr.....
:060730 61 63 68 6F 72 64 00 FF FF FF FF FF 00 00 0D EC achord.....
:060740 61 67 31 00 FF FF FF FF FF FF 00 00 03 B6 ag1.....
:060750 61 67 32 00 FF FF FF FF FF FF 00 00 03 78 ag2.....x
:060760 61 67 33 00 FF FF FF FF FF FF 00 00 03 F4 ag3.....
:060770 61 6C 65 61 64 00 FF FF FF FF FF 00 00 1B 86 ahead.....
:060780 61 6C 6F 6E 67 00 FF FF FF FF FF 00 00 22 F8 along....."
:060790 61 73 6E 61 72 65 00 FF FF FF FF FF 00 00 0B 38 asnare.....8
:0607A0 62 31 62 61 73 73 00 FF FF FF FF FF 00 00 0E B8 blbass.....

```

Figure 9f– Memory dump of file table from disk 2(5)

```

:0607B0 62 31 62 73 64 72 00 FF FF FF FF FF 00 00 0A 64 blbsdr.....d
:0607C0 62 31 6D 61 6A 00 FF FF FF FF FF 00 00 0E 8C blmaj.....
:0607D0 62 31 6D 69 6E 00 FF FF FF FF FF 00 00 0F 92 blmin.....
:0607E0 62 31 6D 72 6D 00 FF FF FF FF FF 00 00 0A 6A blmwn.....J
:0607F0 62 31 70 61 6E 00 FF FF FF FF FF 00 00 12 1C blpan.....
:060800 62 31 73 6E 61 72 65 00 FF FF FF FF 00 00 0A 44 blsnare.....D
:060810 62 31 73 6F 6E 61 72 00 FF FF FF FF 00 00 13 04 blsonar.....
:060820 62 31 74 6F 6D 32 00 FF FF FF FF FF 00 00 09 2E blton2.....
:060830 62 32 62 61 6C 00 FF FF FF FF FF 00 00 09 54 b2bal.....I
:060840 62 32 62 61 6C 32 00 FF FF FF FF FF 00 00 11 E8 b2bal2.....
:060850 62 32 62 61 73 73 00 FF FF FF FF FF 00 00 0F 54 b2bass.....I
:060860 62 32 68 61 6E 64 31 00 FF FF FF FF 00 00 0B F4 b2hand1.....
:060870 62 32 6C 65 61 61 64 00 FF FF FF FF 00 00 10 C0 b2lead.....
:060880 62 32 73 74 72 69 6E 67 00 FF FF FF 00 00 0B 44 b2string.....D
:060890 62 32 74 6F 6D 31 00 FF FF FF FF FF 00 00 0D 9A b2tom1.....
:0608A0 6D 62 64 72 00 FF FF FF FF FF FF 00 00 0A 52 nbdr.....R
:0608B0 6D 63 79 6D 00 FF FF FF FF FF FF 00 00 07 E4 ncym.....ä
:0608C0 6D 64 61 6E 67 65 72 00 FF FF FF FF 00 00 15 C6 ndanger.....
:0608D0 6D 67 74 72 31 00 FF FF FF FF FF 00 00 0C 92 ngtr1.....
:0608E0 6D 67 74 72 32 00 FF FF FF FF FF 00 00 0E 2C ngtr2.....
:0608F0 6D 73 6E 72 00 FF FF FF FF FF FF 00 00 0F 26 nsnr.....&
:060900 6D 73 70 65 65 64 75 70 00 FF FF FF 00 00 0B 88 nspeedup.....
:060910 6D 74 70 31 00 FF FF FF FF FF FF 00 00 11 4A ntp1.....J
:060920 6D 74 70 32 00 FF FF FF FF FF FF 00 00 0B 56 ntp2.....V
:060930 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF

```

Figure 9g– Memory dump of file table from disk 2(6)

```

:060BE0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF .....
:060BF0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF .....
:060C00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
:060C10 01 01 01 01 01 01 01 01 01 01 01 01 01 02 .....
:060C20 02 02 02 02 02 02 02 02 02 02 03 03 03 03 .....
:060C30 03 03 03 03 03 03 03 03 03 03 03 03 04 04 .....
:060C40 04 04 04 04 04 04 04 04 04 04 05 05 05 05 .....
:060C50 05 05 05 05 05 05 05 05 06 06 06 06 06 06 .....
:060C60 06 06 07 07 07 07 07 07 07 07 07 07 07 08 .....
:060C70 08 08 08 08 08 08 08 08 08 08 09 09 09 09 .....
:060C80 09 09 09 09 09 09 09 09 09 09 09 0A 0A 0A .....
:060C90 0A 0A 0A 0A 0A 0A 0A 0B 0B 0B 0B 0C 0C 0C .....
:060CA0 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C .....
:060CB0 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C .....
:060CC0 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C .....
:060CD0 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C .....
:060CE0 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0D 0D 0D .....
:060CF0 0E 0E 0E 0F 0F 0F 0F 10 10 10 11 11 12 12 .....
:060D00 12 12 13 13 13 14 14 15 15 15 15 15 15 16 .....
:060D10 17 17 18 18 19 19 1A 1A 1A 1A 1B 1B 1B 1B .....
:060D20 1C 1C 1C 1C 1C 1D 1D 1D 1D 1E 1E 1E 1F 1F .....
:060D30 1F 20 20 20 21 21 21 21 21 21 22 22 22 23 .....
:060D40 23 23 23 23 24 24 24 24 25 25 26 26 27 28 #####$%&%'<
:060D50 28 28 28 29 29 29 2A 2A 2A 2B 2B 2B 2C 2C ((()))*****+
:060D60 2C 2D 2D 2E 2E 2F 2F 2F 2F 30 30 30 31 31 ,--..///0000111

```

Figure 9h– Memory dump of file table from disk 2(7)

```

:060D70 31 31 32 32 32 32 33 33 33 34 34 34 35 35 36 1122223334445556
:060D80 36 36 36 36 37 37 37 38 38 38 39 39 39 3A 3A 3B 6666777888999::
:060D90 3B 3B 3C 3C 3C 3C 3C 3C 3D 3D 3E 3E 3E 3E 3E 3E <<<<<=>>>>>
:060DA0 3E 3E 3E 3E 3E 3E 3E 3E 3E 3E 3E 3E 3E 3E 3E 3E >>>>>>>>>>>
:060DB0 3E 3E 3E 3E 3E 3E 3E 3E 3F 3F 3F 3F 3F 3F 3F 3F >>>>>>>??????
:060DC0 3F 3F 3F 3F 3F 3F 3F 3F 3F 3F 3F 3F 40 40 40 40 ??????????00000
:060DD0 40 40 40 40 40 40 40 40 40 40 40 40 40 40 40 40 0000000000000000
:060DE0 40 40 41 41 41 41 41 41 41 41 41 41 41 41 41 41 0000000000000000
:060DF0 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 0000000000000000
:060E00 41 41 41 41 42 42 42 42 42 42 43 43 44 44 44 44 AAAABBBBBBCCDDDDDD
:060E10 44 44 44 44 45 45 45 45 46 46 47 47 47 48 48 48 DDDDEEEFFFGGHHHH
:060E20 49 49 49 49 4A 4A 4A 4A 4B 4B 4B 4B 4C 4C 4D 4D IIIIJJJJKKKKLLMM
:060E30 4D 4D 4D 4D 4E 4E 4E 4E 4F 4F 4F 4F 4F 4F 50 50 MMMMNNNN00000000P
:060E40 50 50 50 50 51 51 51 51 52 52 52 52 52 52 52 52 PPPP0000RRRRRRRR
:060E50 53 53 53 53 54 54 54 54 54 55 55 55 55 55 56 56 SSSS$TTTTUUUUUUUV
:060E60 56 56 57 57 57 57 58 58 58 58 59 59 59 59 59 59 VVVVMMWXXXXXXYYY
:060E70 59 5A 5A 5A 5A 5A 5A 5A 5A 5A 5B 5B 5B 5B 5B 5B YZZZZZZZZZZZ[[[[[
:060E80 5B 5B 5B 5B 5C 5C 5C 5C 5C 5C 5C 5D 5D 5D 5D 5D [[[[\[[[[[[[[[[[[
:060E90 5D 5E 5E 5E 5E 5E 5F 5F 5F 60 60 60 60 60 60 60 JAAAAA '[[[[[[
:060EA0 61 61 61 61 61 61 62 62 62 62 62 63 63 63 63 63 aaaaaaa5555555555
:060EB0 63 64 64 65 65 66 66 66 66 66 66 67 67 67 67 67 cddeeffffffggggg
:060EC0 68 68 68 68 68 69 69 69 69 69 69 69 69 69 69 69 hhhhhiiiiiiiiiiii
:060ED0 6A 6A 6A 6B 6B 6B 6B 6B 6B 6B 6C 6C 6C 6C 6C 6C jjjjkkkkkkllllllm
:060EE0 6D 6D 6D 6D 6E 6E 6E 6E 6E 6E 6F 6F 6F 6F 6F 6F mmmmmnnnnnnnnnnnnnn
:060EF0 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F oooooooooooooooooo

```

Figure 9i– Memory dump of file table from disk 2(8)

```

:060F00 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 70 70 70 00000000000000ppp
:060F10 70 70 70 70 70 70 70 70 70 70 70 70 70 70 70 ppppppppppppppppppp
:060F20 70 70 70 70 70 70 70 70 70 70 70 70 70 70 70 ppppppppppppppppppp
:060F30 70 70 70 70 71 71 71 72 72 72 73 73 73 74 75 pppppqqrrrrsssstu
:060F40 76 77 77 77 77 77 77 78 78 78 78 78 78 78 78 vvvvvvvvvvxxxxxxxx
:060F50 78 79 79 79 7A 7A 7A 7A 7B 7B 7B 7C 7C 7C 7D xvvvzzzz{{{|||}}
:060F60 7D 7D 7D 7E 7E 7E 7F 7F 7F 7F 80 80 80 81 81 }}} .....
:060F70 81 81 81 82 82 82 83 83 83 84 84 84 84 84 85 .....
:060F80 85 85 86 86 86 87 87 87 87 87 88 88 88 89 89 .....
:060F90 89 8A 8A 8A 8B 8B 8C 8C 8C 8C 8C 8D 8D 8D 8D .....
:060FA0 8E 8E 8E 8E 8F 8F 8F 8F 90 90 90 91 91 91 91 .....
:060FB0 92 92 92 FF FF FF FF FF FF FF FF FF FF FF FF .....
:060FC0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF .....
:060FD0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF .....

```

Figure 9j. – Memory dump of file table from disk 2(9)

```

sm lemmings_d2_filetable, 60000 61000
Disk ok
-

```

Figure 9k – Save off file table for disk 2.

There are a lot more files this time and if we look at the amount of blocks used \$60c10 to \$60fb3 that is a total of \$3a3 (931) blocks which equals \$ec800 (953344) bytes. We can use the same technique as for disk 1 to extract the contents but we will need to use fast memory this time as we won't have enough free chip memory (we can only load between \$400 - \$60000 otherwise we will overwrite the mfm buffer). We must also change the file name of the first file to be 'main'. This will result in more data than we can fit on one disk. Save the files in two parts onto two separate disks. (See figure 10a, figure 10b and figure 10c).

The two files extracted from disk 2 (length \$74400 and \$74800) total \$e8c00 bytes or 953344. We need to figure out a way we can get this data written back to disk. A couple of possibilities are

- 1) (Re)pack some of the files and attempt to reduce the data to a size that will fit on the disk
- 2) Split the data over two disks resulting in a 3 disk crack (this will pose additional problems as we will need to somehow make the game ask for the correct disk as needed).
- 3) Try to organise the data in such a way that it can still fit on two disks.

In this case repacking the files might be a good idea as creating a 3 disk crack would be the least preferable option. However given the number of files on the disk and the block size being used (\$400) it's entirely possible that if we re-organise the files so that they are entirely back to back on the disk and use a byte based loader that might work. Looking at the file table we see that there are \$92 files. If we assume that the wasted space at the end of each block will be on average half the block size then that equals $92 * \$200 = \12400 . The total size $\$e8c00 - \12400 gives $\$d6800$ or 878592 bytes. This should be enough to fit nicely (in theory). What we need to check now is whether or not this will work in practise.

```

bs 700d0
Breakpoint inserted
Ready.
x
No known virus in memory!
Ready.
Breakpoint raised at address: 000700D0
r
D0=00000000 00000000 000000F0 00000000 00000000 00000F4F FFFFFFFF 0000FFFF
A0=00070610 00000400 00000001 00202EB0 00002E20 000705F0 00DFF000 00000000
PC = 000700D0 USP = 00080000 SR = 0004 T=0 S=0 I=000 X=0 N=0 Z=1 V=0 C=0
r al 280000
D0=00000000 00000000 000000F0 00000000 00000000 00000F4F FFFFFFFF 0000FFFF
A0=00070610 00280000 00000001 00202EB0 00002E20 000705F0 00DFF000 00000000
PC = 000700D0 USP = 00080000 SR = 0004 T=0 S=0 I=000 X=0 N=0 Z=1 V=0 C=0
o 1, 60c10 60del
Ready.
n 60000
:060000 52 65 73 65 72 76 65 64 00 00 00 00 00 00 40 00 Reserved.....0.
:060010 6D 61 69 6E 00 FF FF FF FF FF FF FF 00 07 44 00 main.....D.
:060020 4F 62 6A 65 63 74 73 31 00 FF FF FF 00 00 28 A8 Objects1.....(.
bs 700d4
Breakpoint inserted
Ready.

```

Figure 10a – set up ready to load disk 2 data part 1

```

No known virus in memory!
Ready.
Breakpoint raised at address: 000700D4

sm lemmings_d2_p1, 280000 2f4400
Disk ok

```

Figure 10b – save data from disk 2 part 1

```

n 60000
:060000 52 65 73 65 72 76 65 64 00 00 00 00 00 00 40 00 Reserved.....0.
:060010 6D 61 69 6E 00 FF FF FF FF FF FF FF 00 07 48 00 main.....H.
:060020 4F 62 6A 65 63 74 73 31 00 FF FF FF 00 00 28 A8 Objects1.....(.
o 2, 60c10 60del
Ready.
o 1, 60del 60fb3
Ready.
n 60fb0
:060FB0 01 01 01 FF FF FF FF FF FF FF FF FF FF FF FF FF .....
r al 280000
D0=00000000 00000000 000000F0 00000000 00000000 00000001 FFFFFFFF 00003060
A0=00070610 00280000 00C014E2 00C04780 00001558 000705F0 00DFF000 00C80000
PC = 000700D0 USP = 00080000 SR = 0004 T=0 S=0 I=000 X=0 N=0 Z=1 V=0 C=0
bs 700d4
Breakpoint inserted
Ready.
x
No known virus in memory!
Ready.
Breakpoint raised at address: 000700D4
sm lemmings_d2_p2, 280000 2f4800
Disk ok

```

Figure 10c – extract and save disks from disk 2 part 2

Take a look at figure 11a and see the code I wrote to run through the files and create a new copy of the file data with all of the blank space between removed. This expects the source disk data to be stored at \$200000 and it will create a new copy at \$300000. If we setup and run this code (See Figure 11b). We see in the register a2 it has written up to address \$3d8828 – which means it has written \$d8828 bytes (or 886824) which equates to around 866k. This is good news for us. Write this data out to tracks starting at track 2 to a new blank disk. (See figure 11c). The reason we are starting at track 2 is to give us some space at the start of the disk for other stuff eg. The file table and anything else we might want to store.

Clear out the memory between \$200000 and \$400000 and we can now repeat the same process for disk 1 then we have the data from both disks saved. (See figure 11d, 11e and 11f)

```

~040000 LEA    00060010,A0
~040006 LEA    00200000,A1
~04000C LEA    00300000,A2
~040012 MOVE.L C(A0),D1
~040016 MOVE.B (A1)+,(A2)+
~040018 SUBQ.L #1,D1
~04001A BNE    00040016
~04001C MOVE.L A1,D2
~04001E AND.L  #3FF,D2
~040024 BEQ    0004002E
~040026 ADDA.L #1,A1
~04002C BRA    0004001C
;=====
~04002E LEA    10(A0),A0
~040032 CMPI.L #FFFFFFF,(A0)
~040038 BNE    00040012
~04003A BRA    0004003A
;=====
-

```

Figure 11a – code to remove the blank space from the disk data.

```

ln lemmings_d2_p1, 200000
Loading from 200000 to 274400
Disk ok
ln lemmings_d2_p2, 274400
Loading from 274400 to 2E8C00
Disk ok
ln lemmings_d2_filetable, 60000
Loading from 060000 to 061000
Disk ok
r pc 40000
D0=00000000 00074800 000000F0 00000000 00000000 00000001 FFFFFFFF 00000003
A0=00070610 00280000 00000001 00FE86EE 00001558 000705F0 00DFF000 00C80000
PC = 00040000 USP = 00080000 SR = 0004 T=0 S=0 I=000 X=0 N=0 Z=1 V=0 C=0
bs 4003a
Breakpoint inserted
Ready.
x
No known virus in memory!
Ready.
Breakpoint raised at address: 0004003A
r
D0=00000000 00000000 00000000 00000000 00000000 00000001 FFFFFFFF 00000003
A0=00060930 002E8C00 003D8828 00FE86EE 00001558 000705F0 00DFF000 00C80000
PC = 0004003A USP = 00080000 SR = 0004 T=0 S=0 I=000 X=0 N=0 Z=1 V=0 C=0

```

Figure 11b – setup and run code to remove blank space.

```

x
No known virus in memory!
Ready.
Breakpoint raised at address: 0004003A
r
D0=00000000 00000000 00000000 00000000 00000000 00000001 FFFFFFFF 00000046
A0=00060930 002E8C00 003D8828 00C04780 00001558 000705F0 00DFF000 00C80000
PC = 0004003A USP = 00080000 SR = 0004 T=0 S=0 I=000 X=0 N=0 Z=1 V=0 C=0

wt 2 !158 300000
Disk ok
-

```

Figure 11c – save disk 2 data to a new blank disk tracks starting at track 2.

```

o 0, 200000 400000
Ready.
-

```

Figure 11d – clear memory ready for disk 1.

```

In lemmings_d1 p1, 200000
Loading from 200000 to 25C000
Disk ok
In lemmings_d1 p2, 25C000
Loading from 25C000 to 2B8000
Disk ok
In lemmings_d1 filetable, 60000
Loading from 060000 to 061000
Disk ok
r pc 40000
D0=00000000 00000000 00000000 00000000 00000000 00000001 FFFFFFFF 00000046
A0=000600B0 002B8000 003B6DDC 00C04780 00001558 000705F0 00DFF000 00C80000
PC = 00040000 USP = 00080000 SR = 0004 T=0 S=0 I=000 X=0 N=0 Z=1 V=0 C=0
bs 4003a
Breakpoint inserted
Ready.
x
No known virus in memory!
Ready.
Breakpoint raised at address: 0004003A
r
D0=00000000 00000000 00000000 00000000 00000000 00000001 FFFFFFFF 00000046
A0=000600B0 002B8000 003B6DDC 00C04780 00001558 000705F0 00DFF000 00C80000
PC = 0004003A USP = 00080000 SR = 0004 T=0 S=0 I=000 X=0 N=0 Z=1 V=0 C=0

```

Figure 11e – convert track data for disk 1

```

wt 2 !158 300000
Disk ok

```

Figure 11f – write out track data for disk 12c

Now we can write out the file table to each of the two disks starting at track 1. Before we save we need to update the first **Reserved** file entry to point to the start of track 2 (\$1600 bytes per track on standard AmigaDOS disk so start is \$2c00 for track 2) -see figure 12a and 12b. Next we copy across track 0 (bootblock and the code that gets loaded from the bootblock) onto our cracked disk 1. (See figure 12c).

Insert a new standard load routine

We should now have all the mfm data extracted from the original disks that we need and we can start thinking about patching the load routines to load from our copied disks. As an mfm cracker you should have a load routine of choice. Ideally this load routine will be as compact as possible and as flexible as possible. It should be entirely pc relative and not rely on processor loops for its timings. In this case it will also need to be a byte passed loader (eg you can pass in the start position as a byte offset). It should also not use the blitter for mfm decoding.

My own load routine is 766 bytes and I keep a precompiled binary handy. This tutorial will not walk you through creating your own loader. You can do this as a separate exercise, use one of the ones posted on Flashtro.com or use the one provided here. My loader expects the following:

- load address in a0

- mfm buffer address in a1

- start disk offset in bytes in d0

- length (in bytes) in d1

- drive number in d2

it returns:

- error code in d0 (0 = ok, non-zero for error).

In order to use this we will need to write a small piece of interface code which will set up the registers as passed into the Lemmings loader. What we know of the Lemmings loader so far is:

- Register D1 indicates operation code.

- Operation code 5 is a load file table operation which takes

 - Register d0 - unknown

 - Register a0 – file table load address

 - Register a1 – mfm data buffer area

Operation code 0 is a load file operation which takes

Register a0 – filename

Register a1 – load address.

Returns:

D0: error code (0 = ok)

D1: file size (we must emulate this in our loader otherwise it will fail later on)

We will need to store the file table address and mfm addresses as these are not passed into the main load routine and in addition we will need some code which calculates the start address and length based off of the filename. We will search the file table for the correct file and then pull out the length and then total up the lengths of the preceding files to calculate the start offset. See figures 13a through 13d. Notice the code at \$50058 we stole from the original loader to search the file table and the code at \$50092 that pulls out the file size from the relevant file table entry and then totals up the previous file table entries to determine the load start address. The main load routine itself starts at \$500b6. It is not shown here.

Now we can test our loader by loading in the boot track, executing until a sensible point and then load our new loader over the top of the original loader and see how we go. (See Figure 13e and 13f). We now have code loaded at \$400 so looks like we are good so far.

If we now patch the code on disk we have the start of a cracked disk 1. (See figure 13g). Repeat the steps to test the loader and get a disassembly of the code at \$400 (See figure 13h). Looking at the code the first thing it does is some basic hardware register setup and then copies the code from \$44e to \$78ad0. It then executes a trap to start executing at \$78ad0. Breakpoint at \$44c and then at \$78ad0. Then get a disassembly from \$78ad0 (see figure 13i). Take a quick look at this code then replace the original Lemmings disk in the drive and let it continue on loading. If all works well then we are doing well.

Restart the process all of over again (read track 0 into memory, breakpoint at \$40048, run, breakpoint at \$700f0, run, breakpoint at \$44e, run, breakpoint at \$78ad0, swap disks back to original disk 1, run) then as soon as it starts loading again jump into action replay.

Chances are now you will be in the next loader code. Scroll around and have a look at the code and you may start to notice some familiar things from the previous loader we patched. Eventually you should end up around \$7f570 which should look like the start of the familiar loader. (See figure 13j). This looks to be another copy of the exact same load routine. We should try and patch this one with our new loader. (See figure 14a and 14b). Ensure the cracked disk 1 is in the drive and let it run. Success (See figure 14c)

```

ln lemmings_d1 filetable, 50000
Loading from 050000 to 051000
Disk ok
n 50000
:050000 52 65 73 65 72 76 65 64 00 00 00 00 00 00 2c 00 Reserved.....0.
:050010 6D 61 69 6E 00 FF FF FF FF FF FF FF 00 00 70 00 main.....p.
:050020 50 73 79 67 6E 6F 73 69 73 00 FF FF 00 00 4B C8 Psygnosis.....K.
wt 1 1 50000
Disk ok
-

```

Figure 12a – write out disk one file table.

```

ln lemmings_d2 filetable, 50000
Loading from 050000 to 051000
Disk ok
n 50000
:050000 52 65 73 65 72 76 65 64 00 00 00 00 00 00 2C 00 Reserved.....
:050010 47 72 6F 75 6E 64 31 00 FF FF FF FF 00 00 38 20 Ground1.....0
:050020 4F 62 6A 65 63 74 73 31 00 FF FF FF 00 00 28 A8 Objects1.....(
wt 1 1 50000
Disk ok

```

Figure 12b – write out disk two file table

```

rt 0 1 50000
Disk ok

wt 0 1 50000
Disk ok
-

```

Figure 12c – copy track 0 from the original disk to the cracked disk 1.

```

~050000 MOVEM.L D2-D7/A0-A6,-(A7)
~050004 CMP.W #5,D1
~050008 BEQ 0005001E
~05000A CMP.W #0,D1
~05000E BEQ 00050046
~050010 CMP.W #2,D1
~050014 BEQ 0005003C
~050016 BRA 00050016
;=====
^050018 MOVEM.L (A7)+,D2-D7/A0-A6
~05001C RTS
;=====
^05001E LEA 50060(PC),A2
~050022 MOVE.L A0,(A2)
~050024 MOVE.L A1,4(A2)
~050028 MOVE.L #1600,D0
~05002E MOVE.L #1000,D1
~050034 CLR.L D2
~050036 BSR 000500CC
~05003A BRA 00050018
;=====
^05003C LEA 50060(PC),A2
~050040 MOVEA.L (A2)+,A0
~050042 MOVEA.L (A2)+,A1
~050044 BRA 00050028

```

Figure 13a – loader interface code (1)

```

~050046 LEA 50060(PC),A2
~05004A MOVEA.L A1,A3
~05004C MOVEA.L (A2),A1
~05004E BSR 00050068
~050050 MOVEA.L A3,A0
~050052 CLR.L D2
~050054 MOVEA.L 4(A2),A1
~050058 MOVE.L D1,-(A7)
~05005A BSR 000500CC
~05005C MOVE.L (A7)+,D1
~05005E BRA 00050018
;=====
^050060 NOP
~050062 NOP
~050064 NOP
~050066 NOP
~050068 BSR 0005006E
~05006A BSR 000500A8
~05006C RTS
;=====
^05006E MOVEM.L D2/D6-D7/A0-A1,-(A7)
~050072 MOVE.W #BF,D7
~050076 MOVEQ #0,D1
~050078 MOVEQ #B,D6
~05007A MOVEM.L A0-A1,-(A7)

```

Figure 13b – loader interface code(2)

```

~05007E MOVE.B (A0)+,D2
~050080 CMP.B (A1)+,D2
~050082 BNE 00050092
~050084 TST.B D2
~050086 BEQ 0005008C
~050088 DBF D7,0005007E
~05008C MOVEM.L (A7)+,A0-A1
~050090 BRA 000500A2
;=====
^050092 ADDQ.W #1,D1
~050094 MOVEM.L (A7)+,A0-A1
~050098 LEA 10(A1),A1
~05009C DBF D7,00050078
~0500A0 MOVEQ #0,D1
~0500A2 MOVEM.L (A7)+,D2/D6-D7/A0-A1
~0500A6 RTS
;=====
^0500A8 LEA 50060(PC),A1
~0500AC MOVEA.L (A1),A1
~0500AE MOVEA.L A1,A0
~0500B0 LSL.L #4,D1
~0500B2 ADDA.L D1,A0
~0500B4 MOVE.L C(A0),D1
~0500B8 CLR.L D0
~0500BA SUBA.L #10,A0

```

Figure 13c – loader interface code (3)

```

~0500C0 CMPA.L A0,A1
~0500C2 BGT 000500CA
~0500C4 ADD.L C(A0),D0
~0500C8 BRA 000500BA
;=====
^0500CA RTS
;=====
^0500CC ORT.B #0,D0

```

Figure 13d – loader interface code(4)

```

rt 0 1 40000
Disk ok
d 40044
~040044 JSR -C6(A6)
~040048 TST.L D0
~04004A BEQ 0004005E
bs 40048
Breakpoint inserted
Ready.
g 4000c
No known virus in memory!
Ready.
Breakpoint raised at address: 00040048

In loader_interface, 700e0
Loading from 0700E0 to 07019C
Disk ok
In newload, 7019c
Loading from 07019C to 07049A
Disk ok
bs 700e0
Breakpoint inserted
Ready.
-

```

Figure 13e – testing new loader(1)

```

X
No known virus in memory!
Ready.
Breakpoint raised at address: 000700E0

bs 700d4
Breakpoint inserted
Ready.
X
No known virus in memory!
Ready.
Breakpoint raised at address: 000700D4
d 400
~000400 MOVEA.L #DFF000,A6
~000406 CLR.W 180(A6)
~00040A MOVE.W #7FFF,9A(A6)
~000410 MOVE.W #7FFF,96(A6)
~000416 MOVE.L #FFFFFFF,44(A6)
~00041E MOVE.B #7F,00BFED01
~000426 MOVE.B #7F,00BFDD00
~00042E MOVE.B 00BFED01,D0
~000434 LEA 44E(PC),A0
~000438 LEA 00078AD0,A1

```

Figure 13f – testing new loader(2)

```

rt 0 !10 40000
Disk ok
d 404a0
~0404A0 MOVE.L #1BF00,D0
~0404A6 CLR.L (A0)+
~0404A8 SUBQ.L #1,D0
~0404AA BNE 000404A6
~0404AE LEA 409F0(PC),A5
~0404B2 LEA 00060000,A0
~0404B8 LEA 1000(A0),A1
~0404BC MOVEQ #0,D0
~0404BE MOVEQ #5,D1
~0404C0 BSR 000404E0
~0404C4 LEA 40A10(PC),A0
~0404C8 MOVEA.L #400,A1
~0404CE MOVEQ #0,D1
In loader_interface, 404e0
Loading from 0404E0 to 04059C
Disk ok
In newload, 4059c
Loading from 04059C to 04089A
Disk ok
wt 0 !10 40000
Disk ok
-

```

Figure 13g – update crack disk with new loader

```

d 400
~000400 MOVEA.L #DFF000,A6
~000406 CLR.W 180(A6)
~00040A MOVE.W #7FFF,9A(A6)
~000410 MOVE.W #7FFF,96(A6)
~000416 MOVE.L #FFFFFF,44(A6)
~00041E MOVE.B #7F,00BFED01
~000426 MOVE.B #7F,00BFDD00
~00042E MOVE.B 00BFED01,D0
~000434 LEA 44E(PC),A0
~000438 LEA 00078AD0,A1
~00043E MOVE.L A1,00000084,S
~000442 MOVE.W #37DC,D0
~000446 MOVE.W (A0)+,(A1)+
~000448 DBF D0,00000446
~00044C TRAP #1
~00044E LEA 111E(PC),A5
~000452 MOVEA.L #80000,A7
~000458 LEA DFC(PC),A0
~00045C MOVE.L A0,0000006C,S
~000460 LEA A5A(PC),A1
~000464 MOVE.L A1,00000068,S
~000468 MOVE.W #C028,9A(A6)

```

Figure 13h – main file code at \$400

```

bs 44c
Breakpoint inserted
Ready.
x
No known virus in memory!
Ready.
Breakpoint raised at address: 0000044C
bs 78ad0
Breakpoint inserted
Ready.
x
No known virus in memory!
Ready.
Breakpoint raised at address: 00078AD0
d 78ad0
~078AD0 LEA 797A0(PC),A5
~078AD4 MOVEA.L #80000,A7
~078ADA LEA 7947E(PC),A0
~078ADE MOVE.L A0,0000006C,S
~078AE2 LEA 790DC(PC),A1
~078AE6 MOVE.L A1,00000068,S
~078AEA MOVE.W #C028,9A(A6)
~078AF0 MOVE.W #2100,SR
~078AF4 LEA 00063310,A0
~078AFA MOVE.W #55EF,D0

```

Figure 13i – stepping into code at \$400

```

~07F570 MOVEM.L D2-D7/A0-A6,-(A7)
~07F574 MOVEA.L #DFF000,A6
~07F57A CLR.W 14(A5)
~07F57E MOVE.W 10(A6),16(A5)
~07F584 CLR.B 1(A5)
~07F588 CLR.B 0(A5)
~07F58C CLR.L C(A5)
~07F590 MOVE.W #FFFF,10(A5)
~07F596 SUBQ.W #1,D1
~07F598 BMI 0007F5F0
~07F59C SUBQ.W #1,D1
~07F59E BMI 0007F5EE
~07F5A2 SUBQ.W #1,D1
~07F5A4 BMI 0007F610
~07F5A8 SUBQ.W #1,D1
~07F5AA BMI 0007F5EE
~07F5AE SUBQ.W #1,D1
~07F5B0 BMI 0007F5EE
~07F5B4 SUBQ.W #1,D1
~07F5B6 BMI 0007F604
~07F5BA SUBQ.W #1,D1
~07F5BC BMI 0007F694
~07F5C0 SUBQ.W #1,D1
~07F5C2 BMI 0007F5EE
~07F5C6 SUBQ.W #1,D1

```

Figure 13j – second loader start.

```

rt 0 1 40000
Disk ok
bs 40048
Breakpoint inserted
Ready.
g 4000c
No known virus in memory!
Ready.
Breakpoint raised at address: 00040048
bs 700da
Breakpoint inserted
Ready.
x
No known virus in memory!
Ready.
Breakpoint raised at address: 000700DA
bs 44c
Breakpoint inserted
Ready.
x
No known virus in memory!
Ready.
Breakpoint raised at address: 0000044C

```

Figure 14a – run through loading sequence until code is located at \$78ad0

```

d 7f570
~07F570 MOVEM.L D2-D7/A0-A6,-(A7)
~07F574 MOVEA.L #DFF000,A6
~07F57A CLR.W 14(A5)
~07F57E MOVE.W 10(A6),16(A5)
In loader_interface,7f570
Loading from 07F570 to 07F626
Disk ok
In newload, 7f626
Loading from 07F626 to 07F924
Disk ok

```

Figure 14b – patch second loader with our load routine



Figure14c – success.

Let the loader run as long as it can and eventually it gets stuck and if you jump into the action replace you will see we are stuck in a loop. (See figure `15a, 15b). If you scroll back through the code you will see we are stuck at the start of our load routine with it having detected an unknown operation code of 2 in D1. We now need to take a better look at the original load routine and see what happens when it is called with a value of 2 in D1. Looking at the load routine in figure 13i we see it jumps to \$7f610 in the case of 2 being passed into D1. We can now boot up the original, wait for the intro animation to play and put a breakpoint at \$7f610 to see what's happening.

It's not immediately obvious what's going on here as there isn't much to go on without diving into all of the subroutines. However upon a little examination (figures 15c to 15f) we see that the routine at \$7f684 (which is the one called when the request is made to load the file table) we see that it stores the file table load address at \$4(a5) and our routine pulls out this address then calls the same piece of code repeated 4 times. To my eye this looks like it possibly reloads the file table to the original address \$400 bytes at a time. We can test this theory by putting a breakpoint at \$7f640. Having a look at the memory where the file table should be then running the remainder of the code and looking at the memory again (See figure 15g). When we do this we see that the file table has been reloaded. We can now go back and make some additional changes to our loader to include this functionality. Our load routine now looks something like figure 15h.

If we now repeat the process shown in figure 13i and 14a to patch the second loader with our new load routine (see figure 16e) and the result is that we now manage to make it all the way to the

insert disk 2 prompt (see figure 16f). Although there does appear to be a little bit of corruption so something isn't quite 100%

```

Loading from 07F626 to 07F924
Disk ok
x
No known virus in memory!
Ready.
d
~07F580 BRA      0007F580
=====
~07F582 MOVEM.L (A7)+,D2-D7/A0-A6
~07F586 RTS
=====
~07F588 LEA      7F5BA(PC),A2
=====
-

```

Figure 15a – stuck in an infinite loop

```

~07F570 MOVEM.L D2-D7/A0-A6,-(A7)
~07F574 CMP.W   #5,D1
~07F578 BEQ     0007F588
~07F57A CMP.W   #0,D1
~07F57E BEQ     0007F5A4
~07F580 BRA     0007F580
=====
~07F582 MOVEM.L (A7)+,D2-D7/A0-A6
~07F586 RTS
=====
~07F588 LEA     7F5BA(PC),A2
~07F58C MOVE.L A0,(A2)
~07F58E MOVE.L A1,4(A2)
~07F592 MOVE.L #1600,D0
~07F596 MOVE.L #1000,D1
~07F59E BSR     0007F626
p
D0=0006CF50 00000002 00001F40 0000FFFF 00000005 0000184F 00000000 0000FFFF
A0=0007F498 0006DEF0 00021906 0007986E 0007B54E 000797A0 000FF000 0007FFC8
PC = 0007F580 USP = 00080000 SR = 2100 T=0 S=1 I=001 X=0 N=0 Z=0 V=0 C=0
-

```

Figure 15b – our loader code. Stuck with a value of 2 in D2.

```

Ready.
bs 7f610
Breakpoint inserted
Ready.
x
No known virus in memory!
Ready.
Breakpoint raised at address: 0007F610
d 7f610
~07F610 BSR     0007F9DA
~07F614 BSR     0007F61A
~07F618 BRA     0007F5D2
=====
~07F61A MOVEM.L D0-D1/A1,-(A7)
~07F61E BSR     0007F988

```

Figure 15c – original loader call at \$7f610

```

~07F9DA BCLR    #7,00BFD100
~07F9E2 MOVE.B  2(A5),D7
~07F9E6 BSR     0007FA46
~07F9EA BCLR    D7,00BFD100
~07F9F0 BSR     0007F88C
~07F9F4 RTS

```

Figure 15d – original loader call continued

```

~07F61A MOVEM.L D0-D1/A1,-(A7)
~07F61E BSR     0007F988
~07F622 BSET    #2,00BFD100
~07F62A CLR.W   10(A5)
~07F62E BSR     0007F8B4
~07F632 TST.W   14(A5)
~07F636 BNE     0007F67E
~07F63A MOVEQ   #0,D1
~07F63C MOVEA.L 4(A5),A1
~07F640 MOVE.W  #400,D7
~07F644 BSR     0007F820
~07F648 TST.W   14(A5)
~07F64C BNE     0007F67E
~07F650 MOVEQ   #1,D1
~07F652 LEA     400(A1),A1
~07F656 BSR     0007F820
~07F65A TST.W   14(A5)
~07F65E BNE     0007F67E
~07F662 MOVEQ   #2,D1
~07F664 LEA     400(A1),A1
~07F668 BSR     0007F820
~07F66C TST.W   14(A5)
~07F670 BNE     0007F67E
~07F674 MOVEQ   #3,D1

```

Figure 15e – more original loader disassembly

```

~07F676 LEA     400(A1),A1
~07F67A BSR     0007F820
~07F67E MOVEM.L (A7)+,D0-D1/A1
~07F682 RTS
=====
~07F684 MOVE.L  A0,4(A5)
~07F688 MOVE.L  A1,8(A5)
~07F68C BSR     0007F694
~07F690 BRA     0007F5D2
=====
~07F694 TST.W   D0
~07F696 BMI     0007F6AA
~07F69A ADDQ.W  #3,D0
~07F69C MOVE.B  D0,2(A5)
~07F6A0 BSR     0007F9DA
~07F6A4 BSR     0007F61A
~07F6A8 RTS
=====
~07F6AA MOVEQ   #2,D0

```

Figure 15f – more loader disassembly.

```

r
D0=000600CB 00000000 000000BC FFFF00A0 00000005 0000000F 0000FFFF 0000FF03
A0=0007DF38 00072D10 000340B0 0007986E 00001558 000797A0 000FF000 0007FFB8
PC = 0007F640 USP = 00000000 SR = 2104 T=0 S=1 I=001 X=0 N=0 Z=1 V=0 C=0
n 72d10
:072D10 FF FF 87 FF FF FF FF FA AA 0A B1 67 FF FF FF FF .....g....
:072D20 FF FF 00 00 FF FF FF FF FF FF FF FF .....
:072D30 FF FF FF FF FF FF FF FF FF FF FF FF C3 FF .....
:072D40 FF FF FF BF FF FF F1 0F FF FF FF FF FF 00 00 .....
:072D50 FF FF FF FF FF FF FF FF FF FF FF FF FF FF .....
:072D60 FF FF FF FF FF FF FF FF FF FF C3 FF FF FF FE .....
bs 7f682
Breakpoint inserted
Ready.
x
No known virus in memory!
Ready.
Breakpoint raised at address: 0007F682
n 72d10
:072D10 52 65 73 65 72 76 65 64 00 00 00 00 00 40 00 Reserved.....@.
:072D20 6D 61 69 6E 00 FF FF FF FF FF FF FF 00 00 70 08 main.....p.
:072D30 50 73 79 67 6E 6F 73 69 73 00 FF FF 00 00 4B C8 Psygnosis.....K.
:072D40 64 6D 61 6C 6F 67 6F 00 FF FF FF FF FF 00 00 57 3E dnalogo.....W>
:072D50 72 61 6D 70 69 63 00 FF FF FF FF FF 00 00 20 B4 rampic......
:072D60 6E 6F 69 73 65 73 00 FF FF FF FF FF 00 01 9C 06 noises.....

```

Figure 15g – file table memory dump.

```

~050000 MOVEM.L D2-D7/A0-A6,-(A7)
~050004 CMP.W #5,D1
~050008 BEQ 0005001E
~05000A CMP.W #0,D1
~05000E BEQ 00050046
~050010 CMP.W #2,D1
~050014 BEQ 0005003C
~050016 BRA 00050016
=====
^050018 MOVEM.L (A7)+,D2-D7/A0-A6
~05001C RTS
=====
^05001E LEA 50060(PC),A2
~050022 MOVE.L A0,(A2)
~050024 MOVE.L A1,4(A2)
~050028 MOVE.L #1600,D0
~05002E MOVE.L #1000,D1
~050034 CLR.L D2
~050036 BSR 000500CC
~05003A BRA 00050018
=====
^05003C LEA 50060(PC),A2
~050040 MOVEA.L (A2)+,A0
~050042 MOVEA.L (A2)+,A1
~050044 BRA 00050028

```

Figure 16a – updated loader (1)

```

=====
~050046 LEA      50060(PC),A2
~05004A MOVEA.L A1,A3
~05004C MOVEA.L (A2),A1
~05004E BSR      00050068
~050050 MOVEA.L A3,A0
~050052 CLR.L    D2
~050054 MOVEA.L 4(A2),A1
~050058 MOVE.L   D1,-(A7)
~05005A BSR      000500CC
~05005C MOVE.L   (A7)+,D1
~05005E BRA      00050018
=====
~050060 NOP
~050062 NOP
~050064 NOP
~050066 NOP
~050068 BSR      0005006E
~05006A BSR      000500A8
~05006C RTS
=====
~05006E MOVEM.L D2/D6-D7/A0-A1,-(A7)
~050072 MOVE.W  #BF,D7
~050076 MOVEQ   #0,D1
~050078 MOVEQ   #B,D6

```

Figure 16b – Updated loader(2)

```

~05007A MOVEM.L A0-A1,-(A7)
~05007E MOVE.B  (A0)+,D2
~050080 CMP.B   (A1)+,D2
~050082 BNE     00050092
~050084 TST.B   D2
~050086 BEQ     0005008C
~050088 DBF     D7,0005007E
~05008C MOVEM.L (A7)+,A0-A1
~050090 BRA     000500A2
=====
~050092 ADDQ.W  #1,D1
~050094 MOVEM.L (A7)+,A0-A1
~050098 LEA     10(A1),A1
~05009C DBF     D7,00050078
~0500A0 MOVEQ   #0,D1
~0500A2 MOVEM.L (A7)+,D2/D6-D7/A0-A1
~0500A6 RTS
=====
~0500A8 LEA      50060(PC),A1
~0500AC MOVEA.L (A1),A1
~0500AE MOVEA.L A1,A0
~0500B0 LSL.L    #4,D1
~0500B2 ADDA.L  D1,A0
~0500B4 MOVE.L  C(A0),D1
~0500B8 CLR.L   D0

```

Figure 16c – Updated loader(3)

```

~0500BA SUBA.L  #10,A0
~0500C0 CMPLA.L A0,A1
~0500C2 BGT     000500CA
~0500C4 ADD.L   C(A0),D0
~0500C8 BRA     000500BA
=====
~0500CA RTS
=====
~0500CC ORL.R   #0,D0

```

Figure 16d – Updated loader(4)

```
g 4000c
No known virus in memory!
Ready.
Breakpoint raised at address: 00040048
bs 700da
Breakpoint inserted
Ready.
x
No known virus in memory!
Ready.
Breakpoint raised at address: 000700DA
bs 44c
Breakpoint inserted
Ready.
x
No known virus in memory!
Ready.
Breakpoint raised at address: 0000044C
In loader_interface, 7f570
Loading from 07F570 to 07F63C
Disk ok
In newload, 7f63c
Loading from 07F63C to 07F93A
Disk ok
```

Figure 16e – repeat the loader process



Figure 16f – waiting for disk 2.

What we need to do now is to update the second loader onto the disk that way we can more easily test how things are working. Read in the disk tracks starting at track 2 (where our main file data starts) and start disassembling from here (see figure 17a). We know that our load routine needs to end up address \$7f570 and the start of the relocated data is \$78ad0. We can see that the start of the code to be relocated is at \$4004e based on the data we have read from the disk. So we calculate $\$7f570 - \$78ad0 + \$4004e$ and that's where we should find the load routine. So a quick calculation in action replay and a disassembly shows we are on track (see figure 17b). We can now load our loader routine into memory at the correct point then write back the tracks to disk. (See figure 17c).

Rebooting with our crack disk in the drive should now result in getting as far as the insert disk 2 screen.

```

No known viruses in memory!
Ready.
rt 2 10 40000
Disk ok
d 40000
~040000 MOVEA.L #DFF000,A6
~040006 CLR.W 180(A6)
~04000A MOVE.W #7FFF,9A(A6)
~040010 MOVE.W #7FFF,96(A6)
~040016 MOVE.L #FFFFFFF,44(A6)
~04001E MOVE.B #7F,00BFED01
~040026 MOVE.B #7F,00BFDD00
~04002E MOVE.B 00BFED01,D0
~040034 LEA 4004E(PC),A0
~040038 LEA 00078AD0,A1
~04003E MOVE.L A1,00000084.S
~040042 MOVE.W #37DC,D0
~040046 MOVE.W (A0)+,(A1)+

```

Figure 17a – disassembly of the main code

```

? 7f570-78ad0+4004e
%000000000000001000110101011101110 = $00046AEE = !0000289518
d 46aee
~046AEE MOVEM.L D2-D7/A0-A6,-(A7)
~046AF2 MOVEA.L #DFF000,A6
~046AF8 CLR.W 14(A5)
~046AFC MOVE.W 10(A6),16(A5)
~046B02 CLR.B 1(A5)
~046B06 CLR.B 0(A5)
~046B0A CLR.L C(A5)
~046B0E MOVE.W #FFFF,10(A5)
~046B14 SUBQ.W #1,D1
~046B16 BMI 00046B6E
~046B1A SUBQ.W #1,D1
~046B1C BMI 00046B6C
~046B20 SUBQ.W #1,D1
~046B22 BMI 00046B8E

```

Figure 17b – loader calculation and disassembly.

```

~046B10 BMI 00046B0E
~046B1A SUBQ.W #1,D1
~046B1C BMI 00046B6C
~046B20 SUBQ.W #1,D1
046B22 BMI 00046B8E
In loader_interface, 46aee
Loading from 046AEE to 046BBA
Disk ok
In newload, 46bba
Loading from 046BBA to 046EB8
Disk ok
wt 2 10 40000
Disk ok

```

Figure 17c - load our loader and resave the tracks back to disk.

If we try to insert our disk 2 and click the mouse the disk is not recognised. If we jump into action replay we will most likely see we are executing code in the low area of memory around the \$1000 - \$2000 (see figure 18a). Previously we were executing code around the \$78000 - \$7ffff area of memory so it's a good bet more executable code has been loaded. If we reset and reload our cracked disk 1, then wait for the intro to start playing we can then put a breakpoint at our file load routine to see what's being loaded next. Quit the intro with the mouse and we get a breakpoint hit (see figure 18b).

Looking at what's in the registers and memory we see that a file called **Code** is being loaded to \$400. Put a breakpoint at the end of the load routine and we can investigate what is at \$400 (see figure 18c). Doesn't look like any valid code and nothing obvious in the data. Might be a packed file so we can step through the code and little to see what the next steps are. (See figure 18d). We see that pretty much straight away we end up at a routine that takes the load address and does something then the code is started executing at \$400. A good sign that the routine at \$7932c is a depack routine. Try breakpointing at \$7c690 and we can then get a better view of what's going on at \$400. (See figure 18e) – This looks more promising. If we re-insert the original disk 1 at this stage and let it run everything then loads correctly and the original disk 2 will also allow you to run the game fully.

There is clearly some stuff we need to patch in the code we have just been inspecting. Probably another load routine at the very least. If we want to patch this file we have two options.

- 1) Depack and repack the code with the changes then update the file table respectively
- 2) Patch the code on the fly after it has been depacked prior to the JMP \$400 that we see at \$7c690.

Looking at the depack code it seems as though it is packed with a modified version of the bytekiller packer (with the header placed at the end of the file). It should be possible to unpack and repack but instead of doing that lets look at on the fly patching.

Firstly let's try finding the third loader and patching it in memory to see how we get on. Use the FAQ command to find references to \$dff020 (disk pointer register). See figure 18f. Disassembly backwards from there and we will find the start of the load routine (see figure 18g).

```

d
~0018E6 ANDI.B  #C0,D0
~0018EA BEQ     000018DE
~0018EC RTS
=====
~0018EE MOVEM.L D0-D7/A0-A6,-(A7)
~0018F2 LEA     00009E10,A5
~0018F8 MOVEA.L #DF000,A6
~0018FE MOVE.W  #30,9C(A6)
~001904 ST      20(A5)

```

Figure 18a – example of code being executed while waiting for disk 2.

```

~07F580 CMP.W    #2,D1
~07F584 BEQ      0007F5AC
~07F586 BRA      0007F586
=====
^07F588 MOVEM.L (A7)+,D2-D7/A0-A6
~07F58C RTS
=====
^07F58E LEA      7F5D0(PC),A2
~07F592 MOVE.L  A0,(A2)
~07F594 MOVE.L  A1,4(A2)
~07F598 MOVE.L  #1600,D0
bs 7f5b6
Breakpoint inserted
Ready.

X
No known virus in memory!
Ready.
Breakpoint raised at address: 0007F5B6
v
D0=00000000 00000000 00000000 FFFF00A0 00000005 0000000F 0000FFFF 0000FFFF
A0=00079819 00000400 000340B0 0007986E 00001558 000797A0 00DFF000 0007FFC4
PC = 0007F5B6 USP = 00080000 SR = 2104 T=0 S=1 I=001 X=0 N=0 Z=1 V=0 C=0
n 79819
:079819 43 6F 64 65 00 00 00 00 00 00 00 00 00 00 00 00 Code.....

```

Figure 18b – file loader call after the intro

```

bs 7f588
Breakpoint inserted
Ready.

X
No known virus in memory!
Ready.
Breakpoint raised at address: 0007F588
d 400
~000400 MOVEQ    #FFFFFF80,D7
~000402 ADDI.B   #4,21(A4,D2.L)
~000408 BCLR     #2880,A0
~00040C BTST     #3D79,-(A1)
~000410 NEGX.B   (A4)
~000412 MOVE.L   (A0)+,0000405C.S
~000416 LINEA
n 400
:000400 7E 80 06 34 90 04 2F 21 0E 88 28 80 00 21 3D 79 ~..4../!..(..!=y
:000410 40 14 21 D8 48 5C A0 90 7D 72 00 1A 20 A6 FB 0F 0.!.H\..}r...
:000420 00 43 00 10 35 90 30 09 44 D2 06 0A 52 60 96 AA .C.,5.0.D...R!..
:000430 90 58 4F 36 90 3E 15 E1 22 92 80 58 E2 81 A1 21 .X06.)...".X...!
-

```

Figure 18c – examination of what's at \$400 after load routine is completed.

```

~07F58C RTS
st
D0=00000000 000106C4 00000000 FFFF00A0 00000005 0000000F 0000FFFF 0000FFFF
A0=00079819 00000400 000340B0 0007986E 00001558 000797A0 00DFF000 0007FFFC
PC = 000791CE USP = 00080000 SR = 2100 T=0 S=1 I=001 X=0 N=0 Z=0 V=0 C=0
~0791CE TST.W D0
st
D0=00000000 000106C4 00000000 FFFF00A0 00000005 0000000F 0000FFFF 0000FFFF
A0=00079819 00000400 000340B0 0007986E 00001558 000797A0 00DFF000 0007FFFC
PC = 000791D0 USP = 00080000 SR = 2104 T=0 S=1 I=001 X=0 N=0 Z=1 V=0 C=0
~0791D0 BEQ 00079238
st
D0=00000000 000106C4 00000000 FFFF00A0 00000005 0000000F 0000FFFF 0000FFFF
A0=00079819 00000400 000340B0 0007986E 00001558 000797A0 00DFF000 0007FFFC
PC = 00079238 USP = 00080000 SR = 2104 T=0 S=1 I=001 X=0 N=0 Z=1 V=0 C=0
~079238 RTS
st
D0=00000000 000106C4 00000000 FFFF00A0 00000005 0000000F 0000FFFF 0000FFFF
A0=00079819 00000400 000340B0 0007986E 00001558 000797A0 00DFF000 00080000
PC = 0007C686 USP = 00080000 SR = 2104 T=0 S=1 I=001 X=0 N=0 Z=1 V=0 C=0
~07C686 MOVE.L D1,D0
~07C688 LEA 00000400,S,A0
~07C68C BSR 0007932C
~07C690 JMP 00000400,S

```

Figure 18d – probable depack routine.

```

bs 7c690
Breakpoint inserted
Ready.
x
No known virus in memory!
Ready.
Breakpoint raised at address: 0007C690
d 400
~000400 LEA 41A(PC),A0
~000404 LEA 00009E10,A5
~000408 MOVEA.L #400,A7
~000410 MOVE.L A7,114(A5)
~000414 MOVE.L A0,00000004.S
~000418 TRAP #1
~00041A MOVE.W #2700,SR
~00041E MOVEA.L #DFF000,A6
~000424 MOVE.W #7FFF,9A(A6)
~00042A MOVE.W #7FFF,96(A6)
~000430 MOVE.W #7F00,9E(A6)
~000436 MOVE.W #9500,9E(A6)
~00043C MOVE.L #FFFFFFF,44(A6)
~000444 LEA 0000A3C0,A4
~00044A CLR.B 00BFEC01

```

Figure 18e – depacked code at \$400.

```

faq dff020 400 80000
Search from: 000400 to: 000000
000326 MOVE.L A1,20(A6)
Searched up to adr: 000000
Ready.

```

Figure 18f – looking for the third loader

```

~007FB2 MOVEM.L D2-D7/A0-A6,-(A7)
~007FB6 MOVEA.L #DFF000,A6
~007FBC CLR.W 14(A5)
~007FC0 MOVE.W 10(A6),16(A5)
~007FC6 CLR.B 1(A5)
~007FCA CLR.B 0(A5)
~007FCE CLR.L C(A5)
~007FD2 MOVE.W #FFFF,10(A5)
~007FD8 SUBQ.W #1,D1
~007FDA BMI 00008032
~007FDE SUBQ.W #1,D1
~007FE0 BMI 00008030
~007FE4 SUBQ.W #1,D1
~007FE6 BMI 00008052
~007FEA SUBQ.W #1,D1
~007FEC BMI 00008030
~007FF0 SUBQ.W #1,D1
~007FF2 BMI 00008030
~007FF6 SUBQ.W #1,D1
~007FF8 BMI 000080C6
~007FFC SUBQ.W #1,D1
~007FFE BMI 000080D6
~008002 SUBQ.W #1,D1
~008004 BMI 00008030
~008008 SUBQ.W #1,D1

```

Figure 18g – found the now familiar loader code

```

lm loader_interface, 7fb2
Loading from 007FB2 to 00807E
Disk ok
lm newload, 807e
Loading from 00807E to 00837C
Disk ok
-

```

Figure 18h – load our loader over the third loader code.

```

05 7102
Breakpoint inserted
Ready.
x
No known virus in memory!
Ready.
Breakpoint raised at address: 00007FB2
r
D0=FFFFFFFF 00000005 47726F75 00000004 00000003 00000000 0000222F 0000FFFF
A0=0001E7A2 00058800 00018748 00007004 0000A3C0 00009E10 00DFF000 000003F2
PC = 00007FB2 USP = 00080000 SR = 2100 T=0 S=1 I=001 X=0 N=0 Z=0 V=0 C=0

```

Figure 18i – call to load routine

```

Ready.
Breakpoint raised at address: 0007C690
bs 7fb2
Breakpoint inserted
Ready.
x
No known virus in memory!
Ready.
Breakpoint raised at address: 00007FB2
d 8028
~000028 EXT.L D0
~00002A MOVEM.L (A7)+,D2-D7/A0-A6
~00002E RTS
bs 802e
Breakpoint inserted
Ready.
x
No known virus in memory!
Ready.
Breakpoint raised at address: 0000802E
r
D0=FFFFFFFF 0000FFFF 47726F75 00000004 00000003 00000000 0000222F 0000FFFF
A0=0001E7A2 00058800 00018748 00007004 0000A3C0 00009E10 00DFF000 000003F2
PC = 0000802E USP = 00080000 SR = 2108 T=0 S=1 I=001 X=0 N=1 Z=0 V=0 C=0
-

```

Figure 18j – results of file load

```

000036 BSR 00008164
st
D0=FFFFFFFF 0000FFFF 47726F75 00000000
A0=0001E7A2 00058800 00018748 00007004
PC = 00003C96 USP = 00080000 SR = 2108
~003C96 MOVE.W A0(A5),96(A5)
d 3c80
~003C80 LEA 00058800,A1
~003C86 MOVEQ #5,D1
~003C88 MOVEQ #FFFFFFFF,D0
~003C8A MOVE.L #47726F75,D2
~003C90 JSR 00007FB2
~003C96 MOVE.W A0(A5),96(A5)

```

Figure 18k – unknown data set up in D2 before the call to the load routine.

```

n 3c8a
.003C8A $(GrouN...;n...+n...Nua..fJ0g...a...$A.R.a...A.O.a...$a.ü.a..D

d 8144
~008144 MOVE.L (A7)+,D0
~008146 TST.W 14(A5)
~00814A BNE 0000812A
~00814C TST.L D2
~00814E BMI 0000815E
~008152 MOVEA.L 4(A5),A0
~008156 LEA 10(A0),A0
~00815A CMP.L (A0),D2
~00815C BNE 0000812A
~00815E CLR.W 14(A5)
~008162 RTS
=====
~008164 MOVEM.L D2/D6-D7/A0-A1,-(A7)
~008168 MOVEA.L 4(A5),A1
~00816C MOVE.W #BF,D7

```

Figure 18l

If we now load our loader routines over the top of the existing code at \$7fb2 (See figure 18h) and let it run. There is a problem here – it simply crashes out so somewhere something is wrong. So let's repeat the process again but this time instead of letting it run put a breakpoint at \$7fb2 so we can see where we are going wrong.

First off after setting the breakpoint and letting the code continue we see that there is some drive activity and if you are using WinUAE you will see the drive heads seek to track 0 and stay there for a while then return to their previous position. If you are using a real Amiga you will probably hear the familiar grinding of what sounds like a CopyLock. For now we will ignore this but we will come back to this later. After the track 0 seek we hit the breakpoint and then get a dump of the registers (see figure 18i). At first glance this looks like a standard call to the load filetable routine – however if you breakpoint the same code and run it through on the original disks you will see that the file table load routine fails at this point and returns a nonzero value in D0. (See figure 18j).

If we then look at where the code is called from at this point we see that some value is passed in D2 that we don't really know the meaning of (see figure 18k). The value in D2 looks like ASCII text codes and if you examine them as ASCII you will see they are 'Grou' and if you look at the file table for disk 2 you will see that this is the first 4 characters of the first file (ignoring the reserved file) on disk 2. (See figure 18l). Clearly when a value is passed in D2 the routine expects it to be the first 4 characters of the first file in the file table and it will return an error code in D0 if this is not the case. We now need to update our loader code to comply with this case. See figure 19a for the slightly updated bit of the loader.

```

=====
^050018 MOVEM.L (A7)+,D2-D7/A0-A6
^05001C RTS
=====
^05001E LEA     50074(PC),A2
~050022 MOVE.L  A0,(A2)
~050024 MOVE.L  A1,4(A2)
~050028 MOVEM.L D2/A0,-(A7)
~05002C MOVE.L  #1600,D0
~050032 MOVE.L  #1000,D1
~050038 CLR.L   D2
~05003A BSR     000500E0
~05003E MOVEM.L (A7)+,D2/A0
~050042 TST.L   D2
~050044 BEQ     00050018
~050046 CMP.L   10(A0),D2
~05004A BEQ     00050018
~05004C MOVEQ   #FFFFFF,D0
~05004E BRA     00050018
=====
^050050 LEA     50074(PC),A2
~050054 MOVEA.L (A2)+,A0

```

Figure 19a – updated loader.

```

Ready.
bs 7c690
Breakpoint inserted
Ready.
x
No known virus in memory!
Ready.
Breakpoint raised at address: 0007C690
ln loader_interface, 7fb2
Loading from 007FB2 to 008092
Disk ok
ln newload, 8092
Loading from 008092 to 008390
Disk ok
-

```

Figure 19b – patch third loader in memory.

```

r
D0=00000001 0000FFFF 0000007E 0000FFFF 0000
A0=00000400 00000400 00000400 0007986E 0007
PC = 0007C690 USP = 00080000 SR = 2104 T=0 S

trans 7f570 7fa00 7fb2
Ready.
d 7fb2
~007FB2 MOVEM.L D2-D7/A0-A6,-(A7)
~007FB6 CMP.W #5,D1
~007FBA BEQ 00007FD0
~007FBC CMP.W #0,D1
~007FC0 BEQ 0000800C
~007FC2 CMP.W #2,D1
~007FC6 BEQ 00008002
-

```

Figure 19c – use TRANS command to verify we can copy from loader 2 to loader 3.

If we now repeat the routine going back to breakpoint at \$7c690 after the code has been depacked and then load up our load routine to overwrite the loader at \$7fb2 you should see that it can now recognise our crack disk 2 and starts loading but then it crashes. What we can do now is wire our updated (again) loader into the loader 2 in the same way we did before and that way we can wire in some code to copy the loader 2 over the loader 3 before executing the jmp \$400. (See figures 17a through 17c for the steps to redo loader 2). We can then repeat the same routine going back to breakpoint at \$7c690 after the intro and this time we should be able to use a TRANS command to copy loader 2 over loader3 just to prove that this will work. (See figure 19c).

This seems to work – so now if we do some strategic breakpointing and disk juggling at the right times we should be able to allow the track 0 check to read from the original disk and then still load everything else from our cracked copy. Breakpoint at \$7c690 again and then use TRANS to copy over our loader to loader 3. Put a breakpoint at \$7fb2 so we can re-insert our crack disk at the correct point in time and insert the original disk 1 before allowing the code to continue. Wait for the breakpoint at \$7fb2 and then put the cracked disk 1 into the drive. If all is well at this point it will load right through and you will be able to even play through the game.

CopyLock/Single track check

Now we have successfully extracted all of the data and patched the loader (although we still need to write the code to do the loader 2 to loader 3 memory copy). We can progress onto the CopyLock. Since this will also likely require a patch to the packed code, I am leaving the additional code we need to write until we know how best to bypass the CopyLock. At this stage we don't know for sure that it is a CopyLock, it could be any similar type check.

In order to verify we can breakpoint at \$7c690 again so that all the code is depacked and search for references to address \$24. This this is the trace vector and will definitely indicate whether or not this is a CopyLock or some other similar single track check. What we see in figure 20a is the results of the search. Disassembly of these routines is shown in figures 20b, 20c and 20d. So to my eye the first two look like CopyLock routines and the last one is just random – nothing interesting there.

If we breakpoint at \$4fd8 and \$4fd8 and \$76b8 we can see which of these is being triggered just after the intro. Figure 20e shows the results. If you then disassemble from \$7638 until you find the first non-encrypted code you will see something familiar. (See figure 2f). This is the end of the CopyLock and the start of the loader3 routine. Just to make sure we can set a breakpoint at \$7fb0 and see. (See figure 20g). Quite often when CopyLocks are implemented the first thing you see after the encrypted code is a check of the serial number returned. The coders have not been so silly in this case so we will need to make some effort to find out what the CopyLock is doing. It's also not obvious from what's in the registers after the call what the serial number might be and the other common place for values to be returned from the CopyLock is in the very bottom of the chip memory. We take a dump of this in figure 20h. Nothing much jumps out.

```
No known virus in memory!  
Ready.  
bs 7c690  
Breakpoint inserted  
Ready.  
x  
No known virus in memory!  
Ready.  
Breakpoint raised at address: 0007C690  
faq 24 400 80000  
Search from: 000400 to: 000000  
004FD8 MOVE.L A7,00000024  
004FE2 ADDQ.L #6,00000024  
0076B8 MOVE.L A7,00000024  
0076C2 ADDQ.L #6,00000024  
03E6B4 ADDQ.L #5,00000024.S  
Searched up to adr: 000000  
Ready.
```

Figure 20a – CopyLock?

```

~004F66 MOVEQ    #0,D3
~004F68 PEA      4F74(PC)
~004F6C MOVE.L   (A7)+,00000010
~004F72 ILLEGAL
~004F74 MOVEM.L  D0-D7/A0-A7,-(A7)
~004F78 PEA      4F94(PC)
~004F7C MOVE.L   (A7)+,00000010
~004F82 MOVEA.L  A7,A0
~004F84 LINEF
~004F86 ORI.B    #40,D2
~004F8A ORI.B    #80,SR
~004F8E ORI.B    #7B,D0
~004F92 ORI.B    #48,D2
~004F96 MOVEM.L  4F9A(PC),D0-D7/A0-A6
~004F9C MOVE.L   #4E730000,-(A7)
~004FA2 MOVE.L   #10,-(A7)
~004FA8 MOVE.L   #4DD0B9,-(A7)
~004FAE MOVE.L   #BD96BDAE,-(A7)
~004FB4 MOVE.L   #B386B586,-(A7)
~004FBA MOVE.L   #D046D246,-(A7)
~004FC0 MOVE.L   #246A71F,-(A7)
~004FC6 MOVE.L   #23C17,-(A7)
~004FCC MOVE.L   #42C6F,-(A7)
~004FD2 MOVE.L   #BD96BDAE,-(A7)
~004FD8 MOVE.L   A7,00000024

```

Figure 20b – CopyLock 1?

```

~007648 PEA      7654(PC)
~00764C MOVE.L   (A7)+,00000010
~007652 ILLEGAL
~007654 MOVEM.L  D0-D7/A0-A7,-(A7)
~007658 PEA      7674(PC)
~00765C MOVE.L   (A7)+,00000010
~007662 MOVEA.L  A7,A0
~007664 LINEF
~007666 ORI.B    #40,D2
~00766A ORI.B    #80,SR
~00766E ORI.B    #7B,D0
~007672 ORI.B    #48,D2
~007676 MOVEM.L  767A(PC),D0-D7/A0-A6
~00767C MOVE.L   #4E730000,-(A7)
~007680 ORI.B    #3C,D0
~007684 ORI.B    #10,D0
~007688 MOVE.L   #4DD0B9,-(A7)
~00768E MOVE.L   #BD96BDAE,-(A7)
~007694 MOVE.L   #B386B586,-(A7)
~00769A MOVE.L   #D046D246,-(A7)
~0076A0 MOVE.L   #246A71F,-(A7)
~0076A6 MOVE.L   #23C17,-(A7)
~0076AC MOVE.L   #42C6F,-(A7)
~0076B2 MOVE.L   #BD96BDAE,-(A7)
~0076B8 MOVE.L   A7,00000024

```

Figure 20c - CopyLock 2?

```

~03E684 ORI.? #1200EEA6,-477E(A6)
~03E68C LINEF
~03E68E OR.W D0,SR
~03E690 LINEA
~03E692 SBCD.B D0,D0
~03E694 DIVS.W -236C(A6),D0
~03E698 MOVE.W 48(A6),(A3)+
~03E69C LINEF
~03E69E SUB.W SR,D2
~03E6A0 OR.B -(A4),D1
~03E6A2 MOVE.B (A4)+,(A1)+
~03E6A4 MOVE.B D1,(A1)
~03E6A6 ORI.? #81818181,A2
~03E6AC AND.L D1,D5
~03E6AE AND.L D1,D5
~03E6B0 CMP.W 7E(A4),D4
~03E6B4 ADDQ.L #5,00000024.S
~03E6B8 OR.L D0,D2
~03E6BA MOVE.B A2,(A1)+
~03E6BC LINEA
~03E6BE ADDA.W (A4)+,A6
~03E6C0 OR.B -(A4),D1
~03E6C2 AND.W A0,D5
~03E6C4 ORI.L #12EEDC24,000036EE.S
~03E6CC MOVE.L 000000EE.$,(A2)

```

Figure 20d – CopyLock 3?

```

bs 4fd8
Breakpoint inserted
Ready.
bs 76b8
Breakpoint inserted
Ready.
x
No known virus in memory!
Ready.
Breakpoint raised at address: 000076B8
-

```

Figure 20e – CopyLock results

```

~007F96 ADD.B D2,386E(A2)
~007F9A MOVE.B (A2),(A7)
~007F9C CMP.B -(A2),D7
~007F9E MOVE.W 1027(A6),(A4)+
~007FA2 ROL.B #8,D2
~007FA4 AND.W (A2),D4
~007FA6 MOVE.W 1BA1(A6),(A4)+
~007FAA CMP.W (A2)+,(A0)+
~007FAC MOVEA.W A2,A4
~007FAE NOP
~007FB0 RTS
=====
^007FB2 MOVEM.L D2-D7/A0-A6,-(A7)
~007FB6 MOVEA.L #DFF000,A6
~007FBC CLR.W 14(A5)
~007FC0 MOVE.W 10(A6),16(A5)
~007FC6 CLR.B 1(A5)
~007FCA CLR.B 0(A5)
~007FCE CLR.L C(A5)
~007FD2 MOVE.W #FFFF,10(A5)
~007FD8 SUBQ.W #1,D1
~007FDA BMI 00000032
~007FDE SUBQ.W #1,D1
~007FE0 BMI 00000030
~007FE6 BMI 00000052

```

Figure 20f –end of CopyLock code

```

r
D0=2F3C4E73 00002F3C 00000010 2F3C0004 DDB92F3C BD96BDAE 2F3CB386 B5862F3C
A0=D046D246 2F3C0246 A71F2F3C 00023C17 2F3C0004 2C6F2F3C BD96BDAE 00000388
PC = 000076B8 USP = 00000000 SR = 2100 T=0 S=1 I=001 X=0 N=1 Z=0 V=0 C=0
bs 7fb0
Breakpoint inserted
Ready.
x
No known virus in memory!
Ready.
Breakpoint raised at address: 00007FB0
r
D0=00000000 00000000 70004E75 00003346 4E754E75 00000000 0000222F 0000FFFF
A0=00008AF4 00017244 00018748 00007004 0000A3C0 00009E10 00DFF000 000003F6
PC = 00007FB0 USP = 00000000 SR = 2100 T=0 S=1 I=001 X=0 N=0 Z=0 V=0 C=0
-

```

Figure 20g – after the CopyLock has run.

```

M 0
:000000 00 00 00 00 00 20 00 20 00 00 00 00 00 7E FA .....~
:000010 00 00 03 50 00 FC 08 0E 00 FC 08 20 00 FC 08 22 ...P.ü...ü..ü."
:000020 00 FC 09 0E 00 00 03 34 00 FC 08 28 00 FC 08 2A .ü...4.ü.(.ü.*
:000030 00 FC 08 2C 00 FC 08 2E 00 FC 08 30 00 FC 08 32 .ü.,.ü...ü.ü.2
:000040 4A 39 00 BF E0 01 60 F8 00 FC 08 34 00 FC 08 34 J9...'.ü.4.ü.4
:000050 00 FC 08 34 00 FC 08 34 00 FC 08 34 00 FC 08 34 .ü.4.ü.4.ü.4.ü.4
:000060 00 00 00 00 00 FC 0C 8E 00 00 17 06 00 00 18 EE .....ü.....
:000070 00 FC 0D 6C 00 FC 0D FA 00 FC 0E 40 00 FC 0E 86 .ü.1.ü...ü.ü..
:000080 00 FC 08 36 00 00 04 1A 00 FC 08 3A 00 FC 08 3C .ü.6...ü.:.ü.<
:000090 00 FC 08 3E 00 FC 08 40 00 FC 08 42 00 FC 08 44 .ü.>.ü.ü.ü.B.ü.D
:0000A0 00 00 00 40 00 FC 08 48 00 FC 08 4A 00 FC 08 4C ...ü.ü.H.ü.J.ü.L
:0000B0 00 FC 08 4E 00 FC 08 50 00 FC 08 52 00 FC 08 54 .ü.N.ü.P.ü.R.ü.T
:0000C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
:0000D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
:0000E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
:0000F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
:000100 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....

```

Figure 20h – dump of low memory.

```

Breakpoint inserted
Ready.
x
No known virus in memory!
Ready.
Breakpoint raised at address: 0007C690
bs 7638
Breakpoint inserted
Ready.
bs 7fb0
Breakpoint inserted
Ready.
x
No known virus in memory!
Ready.
Breakpoint raised at address: 00007638
e 9a
;009A 4000 %0100000000101000 ;intena
trans 0 80000 300000
Ready.
x
No known virus in memory!
Ready.
Breakpoint raised at address: 00007FB0

```

Figure 21a – copy all chip memory to fast ram and run CopyLock

```

In make_data, 40000
Loading from 040000 to 040070
Disk ok
d 40000
~040000 MOVEM.L D0-D7/A0-A6, -(A7)
~040004 LEA 00000000, A0
~04000A LEA 00000000, A1
~040010 LEA 00300000, A2
~040016 LEA 00200000, A3
~04001C SF D0
~04001E MOVE.W (A0), D1
~040020 CMP.W (A2), D1
~040022 BEQ 00040038
~040024 TST.B D0
~040026 BNE 00040030
~040028 ST D0
~04002A MOVE.L A0, (A3)+
~04002C MOVEA.L A3, A4
~04002E CLR.W (A3)+
~040030 ADDI.W #1, (A4)
~040034 MOVE.W D1, (A3)+
~040036 BRA 0004003E

```

m 2000

Figure 21b – routine to extract memory changes (1)

```

~040038 TST.B D0
~04003A BEQ 0004003E
~04003C SF D0
~04003E ADDQ.W #2, A0
~040040 ADDQ.W #2, A2
~040042 CMPA.L A0, A1
~040044 BNE 0004001E
~040046 CLR.L (A3)+
~040048 CLR.W (A3)+
~04004A LEA 40070(PC), A0
~04004E MOVE.L A3, (A0)+
~040050 MOVE.L A7, (A0)+
~040052 MOVEM.L (A7)+, D0-D7/A0-A6
~040056 LEA 40070(PC), A7
~04005A MOVEA.L (A7), A7
~04005C MOVEM.L D0-D7/A0-A6, (A7)
~040060 LEA 3C(A7), A0
~040064 LEA 40074(PC), A7
~040068 MOVEA.L (A7)+, A7
~04006A MOVEM.L (A7)+, D0-D7/A0-A6
~04006E BRA 0004006E
;=====
;

```

Figure 21c – routine to extract memory changes (2)

```

bs 4006e
Breakpoint inserted
Ready.
r pc 40000
D0=00000000 00000000 70004E75 00003346 4E754E75 00000000 0000222F 0000FFFF
A0=00008AF4 00017244 00018748 00007004 0000A3C0 00009E10 00DFF000 000003F6
PC = 00040000 USP = 00080000 SR = 2100 T=0 S=1 I=001 X=0 N=0 Z=0 V=0 C=0
x
No known virus in memory!
Ready.
Breakpoint raised at address: 0004006E
r
D0=00000000 00000000 70004E75 00003346 4E754E75 00000000 0000222F 0000FFFF
A0=00008AF4 00017244 00018748 00007004 0000A3C0 00009E10 00DFF000 000003F6
PC = 0004006E USP = 00080000 SR = 2100 T=0 S=1 I=001 X=0 N=0 Z=0 V=0 C=0

```

Figure 21d – running our routine to capture memory changes.

```

:200210 41 F9 00 00 00 00 43 F9 00 08 00 00 45 F9 00 30 A.....C.....E..0
:200220 00 00 47 F9 00 20 00 00 51 C0 32 10 B2 52 67 14 ..G... ..0.2...Rg.
:200230 4A 00 66 08 50 C0 26 C8 28 4B 42 5B 06 54 00 01 J.f.P.&.(KBI.T..
:200240 36 C1 60 06 4A 00 67 02 51 C0 54 48 54 4A B3 C8 6.'J.g.Q.THTJ..
:200250 66 D8 42 9B 42 5B 41 FA 00 24 20 CB 20 CF 4C DF f.B.BIA..$ . .LP
:200260 7F FF 4F FA 00 18 2E 57 48 D7 7F FF 41 EF 00 3C 0.0....WH..A.<
:200270 4F FA 00 0E 2E 5F 4C DF 7F FF 4E 48 00 07 C6 90 0...._LD..NH....
:200280 00 01 4E 48 00 00 00 00 00 00 00 00 00 00 00 ..NH.....
:200290 00 00 70 00 4E 75 00 00 33 46 4E 75 4E 75 00 00 ..p.Nu...3FNUu..
:2002A0 00 00 00 00 22 2F 00 00 FF FF 00 00 8A F4 00 01 ...."/.....
:2002B0 72 44 00 01 87 48 00 00 70 04 00 00 A3 C0 00 00 rD...H..p.....
:2002C0 9E 10 00 DF F0 00 FF FF FF FF FF FF FF FF FF FF ...B.....
:2002D0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF .....
:2002E0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF .....
:2002F0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF .....
:200300 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF .....
:200310 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF .....
:200320 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF .....
:200330 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF .....
:200340 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF .....
sm rnc_data, 200000 2002c6
Disk ok
-

```

Figure 21e – memory differences table

```

In rnc_patch, 40000
Loading from 040000 to 040030
Disk ok
d 40000
~040000 LEA      40030(PC),A0
~040004 TST.L    (A0)
~040006 BEQ      00040016
~040008 MOVEA.L  (A0)+,A1
~04000A MOVE.W   (A0)+,D1
~04000C MOVE.W   (A0)+,(A1)+
~04000E SUB.W    #1,D1
~040012 BNE      0004000C
~040014 BRA      00040004
=====
^040016 ADDQ.L   #6,A0
~040018 LEA      -3C(A7),A7
~04001C MOVEQ    #38,D0
~04001E MOVE.L   0(A0,D0.W),0(A7,D0.W)
~040024 SUB.W    #4,D0
~040028 BPL      0004001E
~04002A MOVEM.L  (A7)+,D0-D7/A0-A6
~04002E RTS
=====
^040030 ADDI.W   #1,(A4)
~040034 MOVE.W   D1,(A3)+

```

Figure 21f – code to replay memory changes.

```

(p) 09 Data1 Electro
*****
No known virus in memory!
Ready.
bs 7c690
Breakpoint inserted
Ready.
x
No known virus in memory!
Ready.
Breakpoint raised at address: 0007C690
trans 7f570 7fa00 7fb2
Ready.
In rnc_patch, 7638
Loading from 007638 to 007668
Disk ok
In rnc_data, 7668
Loading from 007668 to 00792E
Disk ok
-

```

Figure 21g – load up rnc patch and test

```

a 40000
^040000 LEA 0007F570,A0
^040006 LEA 0007FA00,A1
^04000C LEA 00007FB2,A2
^040012 MOVE.W (A0)+,(A2)+
^040014 CMPA.L A0,A1
^040016 BNE 00040012
^040018 LEA 40032(PC),A0
^04001C LEA 40328(PC),A1
^040020 LEA 00007638,A2
^040026 MOVE.W (A0)+,(A2)+
^040028 CMPA.L A0,A1
^04002A BNE 00040026
^04002C JMP 00000400
^040032

In rnc_patch, 40032
Loading from 040032 to 040062
Disk ok
In rnc_data, 40062
Loading from 040062 to 040328
Disk ok

```

Figure 21h – install loader3 and rnc patch code

```

rt 0 !10 50000
Disk ok
trans 40000 40328 52600
Ready.
wt 0 !10 50000
Disk ok

```

Figure 21i – install loader3 and rnc patch code saved to disk 1

```

Ready.
rt 2 !10 40000
Disk ok
d 46ed8
~046ED8 LEA    00078000,A0
~046EDE LEA    00075000,A1
~046EE4 MOVE.L #2600,D0
~046EEA MOVE.L #400,D1
~046EF0 CLR.L  D2
~046EF2 BSR    00046BCE
~046EF6 JMP    00078000
;=====
wt 2 !10 40000
Disk ok
-

```

Figure 21j – found a spare bit of space and put some code in to load our patch installer.

```

faq 400 40000
Search from: 040000 to: C80000
0400A4 LEA    00000400,A0
04054E LEA    00000400.S,A1
0405C4 LEA    00000400.S,A0
043AE8 LEA    00000400.S,A1
043BAA LEA    00000400.S,A1
043BB6 LEA    00000400.S,A0
043BD8 JMP    00000400.S
043BFA LEA    00000400.S,A1
043C06 LEA    00000400.S,A0
043C0E JMP    00000400.S
Searched up to adr: 04C8C0
Ready.

a 43bd8 bra 46ed8
^043BD8
a 43c0e bra 46ed8
^043C0E
^043C12
wt 2 !10 40000
Disk ok
-

```

Figure 21k – jump to our patch routine

So far we are drawing a blank. We know that if we don't run this CopyLock the game stops before the main menu and also there are corrupted graphics at the insert disk 2 code so we need to figure out what it's up to. There are several tutorials on Flashtro on how to patch a CopyLock. I'm going to use a different method than those I've seen demonstrated so far. This method works quite well if you have access to the original disks. It's not feasible to use this if you only have a warp of the disk data for example.

Using the original disks we will breakpoint at \$7638 and \$7fb0. (First breakpointing at \$7c690 as usual as a stepping stone). Then when we hit \$7638 I take a copy of the whole of the chip ram into the top end of my 2mb fast ram. We then disable all interrupts to prevent any interference and run the CopyLock till it ends. I then execute some code wrote to create a table of memory differences in high fast memory. It also saves the contents of all of the registers immediately after the CopyLock call. See figure 21b and 21c for this routine. You can see it compares 0-80000 against \$300000-

\$380000 and creates a difference table at \$200000. (figure 21d) Once this is run we then save the file to disk (figure 21e).

We can then replace the whole CopyLock with some code I wrote to replay the memory and register changes. (see figure 21f). So if we now try this on our cracked disk we should be able to bypass the original disks completely. The usual breakpoint at \$7c690 where we then use TRANS to copy over our loader2 to loader3 and also load up the CopyLock patch routine. Success!

That seems to work. We have successfully patched the CopyLock without even having to figure out what exactly was going on in there.. Hold on though remember there was another CopyLock. I did some further testing to try and trigger this CopyLock and so far have not managed to trigger it. I took a look at the code and it simply checks the returned value directly against a hard coded value. I tried running the CopyLock against both original disks 1 and disk2 and it did not return the correct key code so I can only assume it is a dummy.

So what we need to figure out now is how to hook into the code at \$7c690 and copy over our loader2 to loader3 and patch our CopyLock. Again we could depack the main code file and modify the routines and repack but it's simpler just to patch on the fly. Using my load routine I probably have enough space to tack a mem copy routine onto the end of it but we definitely don't have enough spare space to the rnc patch as well. The easiest way is to use the load routine we already have in memory to pull in a few bytes from the disk and run those before jumping to \$400. The code we want to execute would be something like what we see in figure 21h. We can put this code immediately after the file table on disk 1 (see figure 21i).

Once we have done this all that remains is to plug some code into the spare space we have left on the end of the loader2 to make a call to load our patch code and execute it. See figure 21j and 21k. Notice that there are two places that do a jmp to \$400 that we need to patch. One is when you let the intro run to the end and the other is when you click to exit the intro.

Now we have a crack disk that will run fully and allow you to play the game with no intervention.

As a side note, the CopyLock we have bypassed is a particularly tricky beast in this case. It patches several bits of memory that otherwise will cause problems later on. In addition it stores some data in low memory right at the top of the stack at \$3fe. You can see exactly what bytes it changes by looking at the difference data we built but in the end it is of little consequence since we have bypassed it by replicating its output.

Are we done then?

Not quite. If you play test the game for any length of time you will notice that eventually it will crash or start acting strangely (usually around level 10 for me if I load it up and start playing from the first fun level).

Seems like either we did something wrong or we missed something. Could be any of the following:

- 1) Checksums (very possible)

- 2) Problems with the CopyLock bypass (unlikely given the technique used)
- 3) Corrupt data extracted from the disk (possibly)
- 4) Something else we missed

If its checksums it could potentially be in any of the areas of code we have changed. The most likely place would be in the main code though where we bypassed the CopyLock or the load routine. Next step is to try and rule out a checksum so that we can consider the other points where we may have made a mistake or missed something else vital to the running of the game.

If we load up the game from the original disks then breakpoint at our usual place of \$76c90 have a look at \$7638 where the CopyLock routine lives and \$7fb2 where the loader lives we can find a convenient place to try and make some insignificant changes and see if that triggers anything (see figure 22a). Two things spring to mind. Replacing the MOVEQ #0,D0 with CLR.L D0 (at \$7644) and replace MOVE.L #DFF000,a6 with LEA #DFF000,a6 at \$7fb6. Neither of these changes should affect the running of the code but will generate different opcodes but of the same length of the originals so wont overwrite anything unintentionally and should hopefully trigger something if we are dealing with checksums. When I did this I was able to recreate the crashing problem successfully. Clearly we are dealing with some kind of checksum.

```

x
No known virus in memory!
Ready.
Breakpoint raised at address: 0007C690
d 7638
~007638 MOVE.W #8210,96(A6)
~00763E MOVEM.L 6FEA(PC),D2-D4/D6/A2-A3
~007644 MOVEQ #0,D0
~007646 MOVE.L D0,D1
~007648 PEA 7654(PC)
~00764C MOVE.L (A7)+,00000010
~007652 ILLEGAL
~007654 MOVEM.L D0-D7/A0-A7,-(A7)
~007658 PEA 7674(PC)
~00765C MOVE.L (A7)+,00000010
d 7fb2
~007fb2 MOVEM.L D2-D7/A0-A6,-(A7)
~007fb6 MOVEA.L #DFF000,A6
~007fbc CLR.W 14(A5)
~007fc0 MOVE.W 10(A6),16(A5)
~007fc6 CLR.B 1(A5)
~007fca CLR.B 0(A5)
~007fce CLR.L C(A5)
~007fd2 MOVE.W #FFFF,10(A5)

```

Figure 22a – original CopyLock code and loader code

Checksum(s)

Here's where the fun starts. We know there is at least 1 checksum of some kind but we have little clue where to search for it. I made quite a few attempts at finding it before I finally down the culprit. With these kind of things its often a case of lots of work manually locating the code. I tried various techniques including manually scanning the code for EOR instructions which is often an obvious trick for calculating checksums. I tried manipulating data in memory to try and figure out what kind of

checksum we are dealing with (eg. Try EORing two adjacent bytes, words, longs in memory with the same value to see if it's a simple EOR over the whole memory range). Tried the same technique for ADD and SUB (eg Add a value to one byte, word and long and subtract it from the next). Trying to figure out what will and what won't trigger the routine. None of this got me very far in this case.

I also tried searching for references to \$7638 and \$7fb2 to see if that was the start of the checksummed area and I could find it that way. No joy there. I then tried modifying various random areas of memory in an attempt to figure out what the scope of the checksum was. Everything below our load routine seemed to trigger it. I still had no luck in finding the actual code. I also tried FAQ on address \$400 to see if we could see anything useful there.

I did only see one piece of code that contained an EOR when I started scanning through the code from address \$400. (See figure 22b). It looked slightly suspect but I couldn't figure out what was going on and it didn't look like an obvious checksum routine to me.

```

~00168C NOT.W    D5
~00168E AND.W    D5,(A1)
~001690 OR.W     D2,(A1)
~001692 MOVE.L   D6,D0
~001694 AND.W    D5,0(A1,D0.L)
~001698 OR.W     D3,0(A1,D0.L)
~00169C ADD.L    D6,D0
~00169E AND.W    D5,0(A1,D0.L)
~0016A2 OR.W     D4,0(A1,D0.L)
~0016A6 ADD.L    D6,D0
~0016A8 NOT.W    D5
~0016AA OR.W     D5,0(A1,D0.L)
~0016AE MOVE.W   (A3)+,D2
~0016B0 MOVE.W   48(A5),D5
~0016B4 EOR.W    D2,D5
~0016B6 MOVE.W   D5,48(A5)
~0016BA ADDQ.L   #2,A2
~0016BC SUBQ.W   #1,AC(A5)
~0016C0 BNE      000016E4
~0016C4 SUB.W    00008688,D5
~0016CA MOVE.W   D5,48(A5)
~0016CE MOVE.W   000084C2,AC(A5)
~0016D6 ADD.W    D5,00017244
~0016DC SUB.L    D5,D5
~0016DE BSET     #A,D5

```

Figure 22b – slightly suspect code.

```

P
D0=FFFFFF03 0000FFFF 47726F75 00000F00 00000003 0000FFFF 0000FFFF 0000000F
A0=0001E7A2 00058800 00008E70 00007004 0000A3C0 00009E10 00DFF000 000003F6
PC = 000018E6 USP = 00080000 SR = 2100 T=0 S=1 I=001 X=0 N=0 Z=0 V=0 C=0
d 18de
~0018DE MOVE.B   00BFE001,D0
~0018E4 NOT.B     D0
~0018E6 ANDI.B    #C0,D0
~0018EA BEQ       000018DE
~0018EC RTS

```

Figure 22c – waiting for mouse click

```

d 18ee
~0018EE MOVEM.L D0-D7/A0-A6,-(A7)
~0018F2 LEA 00009E10,A5
~0018F8 MOVEA.L #DFF000,A6
~0018FE MOVE.W #30,9C(A6)
~001904 ST 20(A5)
~001908 ADDQ.W #1,3E(A5)
~00190C BSR 00001D80
~001910 BSR 00001486
~001914 TST.B 30(A5)
~001918 BNE 00001964
~00191C MOVE.W 68(A5),D0
~001920 OR.W 66(A5),D0
~001924 OR.W 64(A5),D0
~001928 BNE 00001934
~00192C SEQ 29(A5)
~001930 BEQ 00001964
~001934 TST.B 2B(A5)
~001938 BNE 00001964
~00193C TST.B 39(A5)
~001940 BNE 00001964
~001944 SUBQ.W #1,68(A5)
~001948 BPL 00001964
~00194C MOVE.W 9A(A5),68(A5)
~001952 SUBQ.W #1,66(A5)

```

Figure 22d – interrupt routine (1)

```

~001956 BPL 00001964
~00195A MOVE.W #3B,66(A5)
~001960 SUBQ.W #1,64(A5)
~001964 MOVEM.L (A7)+,D0-D7/A0-A6
~001968 RTE
;=====
/00196A STP.B 00(A5)

```

Figure 22e – interrupt routine(2)

```

~001D80 LEA 00009DA4,A3
~001D86 BSR 00001D9C
~001D8A TST.B 30(A5)
~001D8E BEQ 00002C7C
~001D92 LEA 2A(A3),A3
~001D96 BSR 00001E2E
~001D9A RTS
;=====
/001D9C MOVE.W A(A6),D1
~001DA0 MOVE.W 6A(A5),D0
~001DA4 ADD.W D0,D0
~001DA6 MOVEA.W D0,A2
~001DA8 LEA 400(A2),A2
~001DAC BSR 00001E82
~001DB0 MOVE.W 6(A3),D1
~001DB4 MOVE.W 8(A3),D2
~001DB8 SUBQ.W #8,D1
~001DBA SUBQ.W #8,D2
~001DBC BSR 000013EE
~001DC0 MOVE.L D0,00014302
~001DC6 MOVE.L D0,0001434A
~001DCC SUBQ.W #8,D1
~001DCE SUBQ.W #8,D2
~001DD0 BSR 00001460
~001DD4 MOVE.L D0,000145EE

```

Figure 22f – interrupt subroutine \$1d80 (1)

```

~001DDA MOVE.L D0,0001480E
~001DE0 BSET #7,D0
~001DE4 MOVE.L D0,00014676
~001DEA MOVE.L D0,00014896
~001DF0 MOVE.L D1,000146FE
~001DF6 MOVE.L D1,0001491E
~001DFC BSET #7,D1
~001E00 MOVE.L D1,00014786
~001E06 MOVE.L D1,000149A6
~001E0C MOVE.L #14302,D0
~001E12 TST.W 10(A3)
~001E16 BEQ 00001E20
~001E1A MOVE.L #1434A,D0
~001E20 LEA 858E(PC),A0
~001E24 MOVE.W D0,4(A0)
~001E28 SWAP D0
~001E2A MOVE.W D0,(A0)
~001E2C RTS
=====

```

Figure 22g – interrupt subroutine \$1d80 (2)

```

=====
~001E82 MOVE.W A(A3),D0
~001E86 MOVE.W (A2),D4
~001E88 MOVE.W D1,A(A3)
~001E8C MOVE.W D0,D2
~001E8E MOVE.W D1,D3
~001E90 ANDI.W #FF,D0
~001E94 ANDI.W #FF,D1
~001E98 SUBQ.W #1,84(A5)
~001E9C SUB.B D1,D0
~001E9E BEQ 00001EB2
~001EA2 BMI 00001EB6
~001EA6 SUB.W D0,6(A3)
~001EAA BPL 00001EB2
~001EAE CLR.W 6(A3)
~001EB2 BRA 00001EEE
=====
~001EB6 NEG.B D0
~001EB8 ADD.W D0,6(A3)
~001EBC TST.B 2F(A5)
~001EC0 BEQ 00001EDE
~001EC4 TST.B 30(A5)
~001EC8 BEQ 00001EDE
~001ECC CMPI.W #8F,6(A3)
~001ED2 BLE 00001EEE

```

Figure 22g – interrupt subroutine \$1e82(1)

```

~001ED6 MOVE.W #8F,6(A3)
~001EDC BRA 00001EEE
/=====
^001EDE CMPI.W #13F,6(A3)
~001EE4 BLE 00001EEE
~001EE8 MOVE.W #13F,6(A3)
~001EEE MOVE.W D2,D0
~001EF0 MOVE.W D3,D1
~001EF2 LSR.W #8,D0
~001EF4 LSR.W #8,D1
~001EF6 LSR.W #2,D4
~001EF8 SUB.W D4,54(A5)
~001EFC TST.W 84(A5)
~001F00 BNE 00001F20
~001F04 MOVE.W #FFFF,6A(A5)
~001F0A MOVE.W 54(A5),D5
~001F0E SUB.W 00009DA0,D5
~001F14 CLR.W 54(A5)
~001F18 ADDA.W D5,A5
~001F1A MOVE.W #3F80,84(A5)
~001F20 ADDQ.W #1,6A(A5)
~001F24 SUB.B D1,D0
~001F26 BEQ 00001F3A
~001F2A BMI 00001F3C
~001F2E SUB.W D0,8(A3)

```

Figure 22h – interrupt subroutine \$1e82(2)

```

All breakpoints deleted!
Ready.
bs 1f0e
Breakpoint inserted
Ready.
x
No known virus in memory!
Ready.
Breakpoint raised at address: 00001F0E
r
D0=000000E0 000000E0 4772E000 0000E000 00000027 000085B4 0000FFFF 0000000F
A0=00008C88 00008754 000082FE 00009DA4 0000A3C0 00009E10 00DFF000 0000038E
PC = 00001F0E USP = 00000000 SR = 2308 T=0 S=1 I=011 X=0 N=1 Z=0 V=0 C=0
d 1f0e
~001F0E SUB.W 00009DA0,D5
st
D0=000000E0 000000E0 4772E000 0000E000 00000027 00000DFF 0000FFFF 0000000F
A0=00008C88 00008754 000082FE 00009DA4 0000A3C0 00009E10 00DFF000 0000038E
PC = 00001F14 USP = 00000000 SR = 2302 T=0 S=1 I=011 X=0 N=0 Z=0 V=1 C=0
~001F14 CLR.W 54(A5)

```

Figure 22i – probably checksum routine.

```
No known virus in memory!  
Ready.  
bs 1f0e  
Breakpoint inserted  
Ready.  
x  
No known virus in memory!  
Ready.  
Breakpoint raised at address: 00001F0E  
d  
~001F0E SUB.W 00009DA0,D5  
st  
D0=00000028 00000028 4772285B 0000285B 00000027 00000AAD 0000FFFF 0000000F  
A0=000008C8 00000875 000008FE 00009DA4 0000A3C0 00009E10 00DFF000 000003A8  
PC = 00001F14 USP = 00080000 SR = 2302 T=0 S=1 I=011 X=0 N=0 Z=0 V=1 C=0  
~001F14 CLR.W 54(A5)
```

Figure 22k – this time it will crash

By this point I was pretty stumped as to how I was going to find the checksum. Eventually I managed to figure out that the game would eventually crash even if I left the game sat waiting at the insert disk 2 screen. This narrowed down the amount of code I needed to sift through considerably since at this point the code is pretty much sat in a tight loop waiting for a mouse click (see figure 22c). That meant that the checksum code had to be in the interrupt routine(s) somewhere.

Using this as a starting point we see that the only interrupts that seem to be active are at \$1706 and \$18ee. I tried disabling the interrupt handler at \$1706 (just making it do an RTE immediately) and it didn't have any effect on the crashing. There doesn't seem to be anything alarming in that code. We now need to focus on the other routine at \$18ee. Looking at the code in figure 22d and 22e there isn't much in the main code. We need to now consider what's at the subroutines at \$1d80 and \$1486. (See figure 22f and 22g). Nothing much here either, we need to delve a bit deeper so we look at the call to \$1e82 (see figure 22g and 22h).

It's all pretty much looking like boring stuff but there is a slight hint of something going off at \$1ef8 through \$1f20. It is keeping a count of something in D2 using SUB and then skipping the following code if \$84(a5) is not 0. We can see above that it subtracts 1 from \$84(a5) and then resets it back to \$3f80 once it reaches 0. The SUB at \$1f0e looks like it could be comparing the result against a known value. If we breakpoint at \$1f0e and wait we should eventually hit this piece of code.

The length of time taken for the code to activate seems roughly along the lines of the time we are seeing for the game to crash so we could be onto a winner here. Looking at the way it adds D5 onto A5 (which is used as a base register for much of the code we have seen so far) it becomes quite clear that if the result in D5 was not zero it could easily cause mayhem by altering A5. When our breakpoint activates we are able to see that the result of the SUB does not end up with D5 being 0. In our case (see figure 22i) we get \$DFE. This doesn't seem to cause an immediate crash if we let it run but that is more luck than anything at this stage. The value in D5 could easily end up being an odd number which would cause an address error on 68000 due to a word memory access on an uneven boundary. If we reset at this point and load the original game to the insert disk 2 screen and then patch the address at \$7fb6 with LEA \$DFF000,a6 and set a breakpoint at \$1f0e again – this time after a bit of a wait we see this time we get \$AAD in D5 which will definitely cause a crash (see figure

22k). We can repeat this test making different changes and we will see that the number varies but if the changes are kept constant the number will not change.

What we cannot do is check to see if the original unmodified code results in a value of 0 as the effect of simply adding a breakpoint at \$1f0e changes the contents of the memory.

If we load up our cracked copy of the game and put a breakpoint at \$1f0e and wait we get a value in D5 of \$5DA6. If we replace the code at \$1f0e with a CLR.W D5 and a couple of NOPs (see figure 23a) we can see what happens when we play the game.

Eventually the game will still crash or start acting strangely as before. It seems as though we might have more than 1 checksum to deal with (or we were mistaken about the code we found). Looking back at the slightly suspect code we found previously (figure 22b). There seems to be some bits in common with the routine we disabled at \$1f0e. The code at \$16bc subtracts 1 from a value and skips over the code at \$16c4 which looks suspiciously like the code at \$1f0e. Especially when we look at the code at \$16ce and then see that the contents of \$84c2 contains \$3f80 it all starts to look pretty familiar. \$3f80 is the number of words it will process in the checksum and once it's done then all it checks the result and starts all over again.

If we do a search for those bytes (\$3f and \$80) we find several memory addresses at which to investigate further. (See figure 23b). Starting to investigate these \$48c doesn't seem like another checksum (probably just setting up the first run of the checksum at the start). \$1f1c and \$84c2 we already know about. Looking at the other addresses returned we don't see anything much of interest.

Now that we know how this checksum routine roughly works (if you haven't understood the code around \$1da0 sets up the current address to read from starting at \$400). The code at \$1e86 reads a word from the memory and the code at \$1efc does the calculation at the end and resets the offset and number of words to process ready for another run. This routine is called once per frame so that it checksums one word of memory each frame until all the range of memory is complete. That is why it takes a while for the checksum to activate.

Based off of this we can surmise that there could also be checksums that work on a byte or longword basis. So we can search for $\$3f80 / 2$ (\$1fc0) for longword checksums and $\$3f80 * 2$ (\$7f00) for byte based checksums. Figure 23c shows the results of the search for longword checksums. The addresses that come back don't seem to contain much in the way of valid code but one of the previous routines we saw reset the data using an absolute address so it's worth doing an FAQ to see if we can find anything interesting (see figure. 23d). With some search around the values seen I managed to track down that I think are 5 checksum routines that need patching at the following addresses:

\$1f0e (the first one we identified)

\$16c4 (the original suspicious code we saw)

\$4efe (this one references the address \$9d3a we saw in figure 23d)

\$3fc8 (this is a longword one that references \$9a9a we saw in figure 23d)

\$3b4c (a second longword one that references \$9eb0 we saw in figure 23d).

I wasn't able to detect any byte based checksum routines.

We can now disable all of these checksums with the following:

1f0e - clr.w d5,nop,nop

16c4 - clr.w d5, nop,nop

4efe - clr.w d0, nop,nop

3fc8 - clr.l d6, nop,nop

3b4c - clr.l d3, nop,nop

After doing this I was able to play the game for around 45 levels without noticing any negative side effects. I then tried to enter a level code and restart the game and it crashed. Seems like we have at least one more checksum to find. All of the routines we have found so far end with one of two possible codes:

SUB(.w/l) absolute_addr,Dn

SUB(.w/l) offset(a5),Dn

See figure 23e and 23f for a list of all the possible combinations of instructions of this type. We can get a list of these opcodes and search for them using the F command. (see figure 23g for the memory dump of these instructions). So we need to search for the following opcodes:

\$9079, \$9279, \$9479, \$9679, \$9879, \$9a79, \$9c79, \$9e79 (sub.w addr, Dn)

\$906d, \$926d, \$946d, \$966d, \$986d, \$9a6d, \$9c6d, \$9e6d (sub.w addr(a5),Dn)

\$90b9, \$92b9, \$94b9, \$96b9, \$98b9, \$9ab9, \$9cb9, \$9eb9 (sub.l addr,Dn)

\$90ad, \$92ad, \$94ad, \$96ad, \$98ad, \$9aad, \$9cad, \$9ead (sub.l addr(a5),Dn)

Using the F command to search for all of these opcodes between \$400 and \$80000 we find a couple of interesting pieces of code (see figure 23h and figure 23i). The routine at \$2b64 shouldn't affect us as it starts at \$4348 and runs through the memory up to \$6348 and we haven't changed anything here so we can leave this be. The second routine looks more bothersome. It starts by moving a6 to a3 and subtracting \$dfec00. This results in \$400 in a3 so we can see this one starts at \$400 and the length is the same as in previous routines (-1 to account for the use of DBF). We will need to replace the SUB at \$30c4 with a CLR.L d0, NOP, NOP as per the previous routines.

If we update our patch code to neutralise all of these checksum checks we end up with the final patch code shown in figure 24a and 24b. Write this code to our cracked disk 1 and you should be able to play the game with no problems.

I've played this through around 40 odd levels without cheating and played every single level and through to the final congratulations screen by training the number of lemmings required to complete each level down to 0 and I've not seen any sign of further checksums.

```
bs 1f0e
Breakpoint inserted
Ready.
x
No known virus in memory!
Ready.
Breakpoint raised at address: 00001F0E
d
~001F0E SUB.W 00009DA0,D5
~001F14 CLR.W 54(A5)
st
D0=FFFF000D 5501000D 47720D2D 00000D2D 00003030 00005DA6 0000FFFF 0000000F
A0=0001E7A2 00058800 000082FE 00009DA4 0000A3C0 00009E10 00DFF000 000003A8
PC = 00001F14 USP = 00000000 SR = 2302 T=0 S=1 I=011 X=0 N=0 Z=0 V=1 C=0
~001F14 CLR.W 54(A5)
r d5 0
D0=FFFF000D 5501000D 47720D2D 00000D2D 00003030 00000000 0000FFFF 0000000F
A0=0001E7A2 00058800 000082FE 00009DA4 0000A3C0 00009E10 00DFF000 000003A8
PC = 00001F14 USP = 00000000 SR = 2302 T=0 S=1 I=011 X=0 N=0 Z=0 V=1 C=0
a 1f0e
^001F0E clr.w d5
^001F10 nop
^001F12 nop
^001F14
```

Figure 23a – patched checksum

```
f 3f 80, 400 80000
Search from: 000400 to: 080000
00048C 001F1C 0004C2 014844 014865 015974 015976 015A40 01E5EB 023089
023668 05BF21 05E96D 068279 072677
Ready.
-
```

Figure 23b – search for \$3f \$80

```

f 1f c0, 400 80000
Search from: 000400 to: 000000
0004E8 009D3A 009E9A 009EA6 009EB0 009EBE 015440 015978 015A3E 0164B8
01AEE6 01B474 01E7A0 02200B 0224D3 02299B 02C882 02C8AA 02D3B0 02D3D8
02E7C2 02E7EA 02F2F0 05E43C 066803 067A6D 070ED6
Ready.
d 4e6
~0004E6 MOVE.W #1FC0,A0(A5)
~0004EC ST 32(A5)
~0004F0 BSR 00003B9C
d 9d30
~009D30 BCS 00009DA0
~009D32 BCC 00009DA7
~009D34 BSR 00009DA3
~009D36 MOVEQ #6C,D0
~009D38 BCS 0000BCFA
d 9e98
~009E98 ORI.B #C0,D0
~009E9C ORI.W #D0,(A0)
~009EA0 ORI.B #0,D0
~009EA4 ORI.B #C0,D0
~009EA8 ORI.B #31,D0
d 9ea4
~009EA4 ORI.B #C0,D0
~009EA8 ORI.B #31,D0

```

Figure 23c – search for \$1f c0 – searching for longword based checksums.

```

Search from: 000400 to: 000000
002B80 MOVE.Wo 00009D3A,AE(A5)
004F12 MOVE.W 00009D3A,AE(A5)
Searched up to adr: 000000
Ready.
faq 9e9a 400 80000
Search from: 000400 to: 000000
001C26 MOVE.W 0001E7A0,8A(A5)
003FB2 SUBQ.W #1,8A(A5)
003FBC MOVE.W 0001E7A0,8A(A5)
Searched up to adr: 000000
Ready.
faq 9ea6 400 80000
Search from: 000400 to: 000000
003AFA MOVE.W 96(A5),D4
003B96 MOVE.W D4,96(A5)
003C96 MOVE.W A0(A5),96(A5)
Searched up to adr: 000000
Ready.
faq 9eb0 400 80000
Search from: 000400 to: 000000
0004E6 MOVE.W #1FC0,A0(A5)
003B58 ADD.W A0(A5),D4
003C96 MOVE.W A0(A5),96(A5)
-

```

Figure 23d – FAQ search for longword checksums.

```

~07F000 SUB.W 00000000,D0
~07F006 SUB.W 00000000,D1
~07F00C SUB.W 00000000,D2
~07F012 SUB.W 00000000,D3
~07F018 SUB.W 00000000,D4
~07F01E SUB.W 00000000,D5
~07F024 SUB.W 00000000,D6
~07F02A SUB.W 00000000,D7
~07F030 SUB.W 2(A5),D0
~07F034 SUB.W 2(A5),D1
~07F038 SUB.W 2(A5),D2
~07F03C SUB.W 2(A5),D3
~07F040 SUB.W 2(A5),D4
~07F044 SUB.W 2(A5),D5
~07F048 SUB.W 2(A5),D6
~07F04C SUB.W 2(A5),D7
~07F050 SUB.L 00000000,D0
~07F056 SUB.L 00000000,D1
~07F05C SUB.L 00000000,D2
~07F062 SUB.L 00000000,D3
~07F068 SUB.L 00000000,D4
~07F06E SUB.L 00000000,D5
~07F074 SUB.L 00000000,D6
~07F07A SUB.L 00000000,D7
~07F080 SUB.L 2(A5),D0

```

Figure 23e – list of instructions we need to search for (1)

```

~07F084 SUB.L 2(A5),D1
~07F088 SUB.L 2(A5),D2
~07F08C SUB.L 2(A5),D3
~07F090 SUB.L 2(A5),D4
~07F094 SUB.L 2(A5),D5
~07F098 SUB.L 2(A5),D6
~07F09C SUB.L 2(A5),D7
~07F0A0 ORI.B #0,D0

```

Figure 23f – list of instructions we need to search for(2)

```

n 7f000
:07F000 90 79 00 00 00 00 92 79 00 00 00 00 94 79 00 00 .y.....y.....y..
:07F010 00 00 96 79 00 00 00 00 98 79 00 00 00 00 9A 79 ...y.....y.....y
:07F020 00 00 00 00 9C 79 00 00 00 00 9E 79 00 00 00 00 ....y.....y....
:07F030 90 6D 00 02 92 6D 00 02 94 6D 00 02 96 6D 00 02 .n...n...n...n..
:07F040 98 6D 00 02 9A 6D 00 02 9C 6D 00 02 9E 6D 00 02 .n...n...n...n..
:07F050 90 B9 00 00 00 00 92 B9 00 00 00 00 94 B9 00 00 .....
:07F060 00 00 96 B9 00 00 00 00 98 B9 00 00 00 00 9A B9 .....
:07F070 00 00 00 00 9C B9 00 00 00 00 9E B9 00 00 00 00 .....
:07F080 90 AD 00 02 92 AD 00 02 94 AD 00 02 96 AD 00 02 .....
:07F090 98 AD 00 02 9A AD 00 02 9C AD 00 02 9E AD 00 02 .....
:07F0A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
:07F0B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
:07F0C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....

```

Figure 23g – memory dump of instructions to search for

```

d 2b64
~002B64 LEA      00004348.S,A0
~002B68 MOVE.W  #FFF,D0
~002B6C MOVEQ   #40,D1
~002B6E MOVE.W  (A0)+,D2
~002B70 MOVE.L  D1,D3
~002B72 MULU.W  D2,D3
~002B74 ADD.L   D3,D1
~002B76 DBF     D0,00002B6E
~002B7A SUB.L   00014A2E,D1
~002B80 MOVE.W  00009D3A,AE(A5)
~002B88 LEA     00014EA4,A0
~002B8E MOVEQ   #1A,D0
~002B90 MULU.W  #60,D0
~002B94 ADD.W   D1,D0
~002B96 ADDA.W  D0,A0
~002B98 MOVEQ   #2F,D0

```

Figure 23h – more possible checksums (1)

```

d 30aa
~0030AA MOVEA.L A6,A3
~0030AC SUBA.L  #DFEC00,A3
~0030B2 MOVE.W  #1FBF,D1
~0030B6 MOVE.B  #0,CCR
~0030BA MOVEQ   #0,D0
~0030BC MOVE.L  (A3)+,D2
~0030BE ADDX.L  D2,D0
~0030C0 DBF     D1,000030BC
~0030C4 SUB.L   000142FE,D0
~0030CA ADD.W   D0,00003286
~0030D0 MOVEQ   #8,D1
~0030D2 MOVEQ   #0,D0
~0030D4 MOVE.L  A0,-(A7)
~0030D6 ADD.B   (A0)+,D0
~0030D8 DBF     D1,000030D6

```

Figure 23i – more possible checksums (2)

```

d 42600
~042600 MOVE.W  #4245,D1
~042604 MOVE.L  #4E714E71,D2
~04260A LEA     000016C4,A0
~042610 MOVE.W  D1,(A0)+
~042612 MOVE.L  D2,(A0)+
~042614 LEA     00001F0E,A0
~04261A MOVE.W  D1,(A0)+
~04261C MOVE.L  D2,(A0)+
~04261E LEA     00004EFE,A0
~042624 MOVE.W  #4240,(A0)+
~042628 MOVE.L  D2,(A0)+
~04262A LEA     00003FC8,A0
~042630 MOVE.W  #4286,(A0)+
~042634 MOVE.L  D2,(A0)+
~042636 LEA     00003B4C,A0
~04263C MOVE.W  #4283,(A0)+
~042640 MOVE.L  D2,(A0)+
~042642 LEA     000030C4,A0
~042648 MOVE.W  #4280,(A0)+
~04264C MOVE.L  D2,(A0)+
~04264E LEA     0007F570,A0
~042654 LEA     0007FA00,A1
~04265A LEA     00007FB2,A2
~042660 MOVE.W  (A0)+,(A2)+

```

Figure 24a – final patch code (1)

```

~042662 CMPA.L  A0,A1
~042664 BNE     00042660
~042666 LEA     42680(PC),A0
~04266A LEA     42976(PC),A1
~04266E LEA     00007638,A2
~042674 MOVE.W  (A0)+,(A2)+
~042676 CMPA.L  A0,A1
~042678 BNE     00042674
~04267A JMP     00000400
-----

```

Figure 24b – final patch code(2)

Done

All that remains is to insert an intro (plenty of room left on disk 1) and a trainer if you so feel.

CONGRATULATIONS!

We made it through... that was a long one... see you next time

Phantasm....