

Amiga cracking

"Alien World"

(c) 1992 Hi-Tec

Things you will need:

1. The original of "Alien World" (c) 1992 Hi-Tec. The CAPS (SPS) version (number 1398) is the one I've used here.
2. An Amiga or a copy of UAE.
3. Action Replay 3 cartridge.
4. Asm-One, or your personal favourite assembler.

Today we're going to look at a horizontal shoot-em-up called "Alien World", that I must admit I don't remember at all from back in the day. It's going to be quite an easy crack, in fact this tutorial was so short that I decided to also crack the Atari ST version of the game (big mistake!). If you are interested, please check out the second part of this document. Anyway, back on-topic! The Amiga cracking tutorial law, Chapter 13 section 337, states that all tutorials must have a screenshot of an attempt to copy the disk with XCopy:



Illustration 1: XCopy Alien World

A custom format disk - good! This gives us something to do at least, lets load the first track into memory using AR and check out the bootblock:

```
>>> rt 0 1 10000
```

```
>>> d 1000c
```

```

MOVEA.L    $00000004.S,A6
MOVE.L     #$7E000,$28(A1)      ;load address
MOVE.L     #$400,$2C(A1)        ;disk offset
MOVE.L     #$E00,$24(A1)        ;length
JSR        -$1C8(A6)            ;DoIO()
JMP        $0007E000            ;jump to loaded code

```

We have no need for this code, since we already read in the entire track (which is \$1600 bytes long) we can just move the code into position with AR:

```
>>> trans 10400 10400+e00 7e000
```

```
>>> d 7e000
```

```

~07E030 MOVE.L #200,D0          LOAD ADDRESS FOR FILETABLE
~07E036 MOVE.L #600,D1          MFM BUFFER
~07E03C MOVE.L D1,D2
~07E03E BSR     0007E1B0          SET VARS FOR LOADER
~07E042 BSR     0007E242          LOAD FILETABLE
~07E046 LEA     7E87A(PC),A0      FILENAME PTR "anim.prg",0
~07E04A MOVE.L #40000,D0         LOAD ADDRESS
~07E050 BSR     0007E1CA          CALL LOADER
~07E054 JSR     00040000          JSR TO LOADED CODE
~07E05A LEA     00DFF000,A6
~07E060 LEA     7E796(PC),A0      PTR TO COPPERLIST
~07E064 MOVE.L A0,80(A6)         SET COP1LCH
~07E068 TST.W   88(A6)           STROBE COPJMP1 TO RESTART COPPER
~07E06C BSR     0007E11C          WAIT FOR VBL
~07E070 LEA     7E88D(PC),A0      FILENAME PTR "load.dat",0
~07E074 MOVE.L #1F40,D0
~07E07A BSR     0007E0AA
~07E07E LEA     7E883(PC),A0      FILENAME PTR "intro.prg",0
~07E082 MOVE.L #20000,D0         LOAD ADDRESS
~07E088 BSR     0007E1CA          CALL LOADER
~07E08C BSR     0007E11C          WAIT FOR VBL
~07E090 JSR     00020000          JSR TO LOADED CODE
~07E096 LEA     7E896(PC),A0      FILENAME PTR "alien.prg",0
~07E09A MOVE.L #37820,D0         LOAD ADDRESS
~07E0A0 BSR     0007E1CA          CALL LOADER
~07E0A4 JMP     00037820          JMP TO MAIN GAME EXE

```

Illustration 2: Code relocated by bootblock to \$07E000

So after going into supervisor mode and turning off DMA etc., the real code begins at \$07E030. I've detailed in the picture above what everything means, but I'll quickly go over how I discovered all this.

If we examine the code at \$07E046 to \$07E054, and \$07E07E to \$07E090, we can clearly see that D0 must be the load address because currently there is nothing at the address it is about to JSR to. This also means that \$07E1CA must be the loader, since the load address is passed to it and when it returns magically there is something there! Since we know D0 is the load address, that must mean A0 is something else... if we examine the address it points to before the first loader call (\$07E87A) we see the null-terminated string "anim.prg", therefore A0 must be a pointer to the filename. This means that somewhere in memory there must be a filetable which includes filenames, but currently we don't know where that resides.

At this point we could disassemble all the loader code and see what it does, or we could reboot and let the game start loading then enter AR and have a look for the filetable. Since I'm lazy I took the second option, and simply reset and let the game begin loading then pressed the magic button and scanned RAM for ascii using the "nq" command:

```
nq c0
000200 DOS1.Alien.a.K.2.anim.prg.e.)0intro.prg.L.$pload.dat.a.alien.np.y.8.sfx.r
00020D aw.}.I@alien.prg.0.ship.dat.U%.panel.dat.\).-.title.dat.h.E5lev1.prg.zW.'
000338 lev1_1.dat.a.lev1_2.dat.&.lev1_3.dat.5.lev1_4.dat.u.+hlev2.prg. hlev2_1.d
000390 at.i.W.lev2_2.dat.M.*plev2_3.dat.i.wlev2_4.dat.5rlev3.prg.lev3_1.dat.D.le
Quick-dump up to address:0003E9
Ready.

m 200
:000200 44 4F 53 31 00 00 B8 00 41 6C 69 65 6E 00 00 00 DOS1....Alien...
:000210 00 00 FF FF FF FF FF FF FF FF FF FF FF FF FF FF .....
:000220 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF .....
:000230 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF .....
:000240 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF .....
:000250 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF .....
:000260 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF .....
:000270 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF .....
:000280 FF FF FF FF FF FF FF FF 03 00 00 00 00 00 01 03 .....
:000290 61 CF 4B E7 89 A5 32 0D 61 6E 69 6D 2E 70 72 67 a.K...2.anim.prg
:0002A0 00 00 80 01 65 01 29 4F 69 6E 74 72 6F 2E 70 72 ....e.)0intro.pr
:0002B0 67 00 80 4C 8D 04 53 DF 6C 6F 61 64 2E 64 61 74 g..L..$pload.dat
:0002C0 00 00 81 61 8F 00 B2 A8 61 6C 69 65 6E 2E 6E 70 ...a....alien.np
:0002D0 00 00 81 8E 79 01 38 D9 73 66 78 2E 72 61 77 00 ....y.8.sfx.raw.
:0002E0 00 00 81 DD 7D 01 49 40 61 6C 69 65 6E 2E 70 72 ....}.I@alien.pr
```

Illustration 3: Filetable at \$000200

Well that explains 2 things - the address \$200 points to some text, with the filetable starting around \$290. If we check where the mfm buffer is pointed to ("e 20" to show \$DFF020/DSKPTH) we see the value \$600.

If you look back at the picture of the code loaded by the bootblock, the first 2 lines are clearly setting up loader variables to point to the load address for the filetable (d0=\$200) and the mfm buffer ptr (d1=\$600).

Now that we know what all the loader arguments are, it's time to rip some files!

There is no need to over-complicate anything here, a quick patch installed by a modified bootblock and patching the instruction at \$07E046 to JMP to our patch will accomplish everything we need. By the time \$07E046 is being executed, the filetable/disk-init stuff has been executed so this is the obvious place to hook. Insert a copy of the original, and lets start patching:

```
>>> rt 0 1 10000
>>> a 10446, "JMP 7F000" <enter>, <esc>
>>> a 1002c <add code as per screenshot below>
>>> bootchk 10000
>>> wt 0 1 10000
```

The new bootblock looks like this:

d 1000c		
~01000C	MOVEA.L	00000004,S,A6
~010010	MOVE.L	#7E000,28(A1)
~010018	MOVE.L	#400,2C(A1)
~010020	MOVE.L	#E00,24(A1)
~010028	JSR	-1C8(A6)
~01002C	MOVE.W	#F0,00DFF180
~010034	BTST	#6,00BFE001
~01003C	BNE	00010034
~01003E	LEA	10060(PC),A0
~010042	LEA	0007F000,A5
~010048	MOVE.W	#20,D0
~01004C	MOVE.L	(A0)+(A5)+
~01004E	DBF	D0,0001004C
~010052	MOVE.W	#F00,00DFF180
~01005A	JMP	0007E000
		DEST ADDR DISK OFFSET LENGTH
~010060	LEA	00000298,A0
~010066	LEA	00000438,A2
~01006C	CMPA.L	A0,A2
~01006E	BNE	0001007C
~010070	MOVE.W	00DFF006,00DFF180
~01007A	BRA	00010070
		SCREEN TO GREEN WAIT LMB (SO WE CAN INSERT ORIG)
~01007C	MOVE.L	#20000,D0
~010082	JSR	0007E1CA
~010088	MOVE.W	#F,00DFF180
~010090	BTST	#6,00BFE001
~010098	BNE	00010090
~01009A	LEA	10064(PC),A0
~01009E	ADDI.W	#10,(A0)
~0100A2	MOVE.W	#F00,00DFF180
~0100AA	BRA	00010060
		RELOC RIPPING CODE TO \$7F000
		SCREEN TO RED JUMP TO LOADED CODE
		PTR TO CURRENT FILENAME END OF FILETABLE DID WE RIP ALL FILES YET? IF NOT, RIP NEXT FILE FLASH SCREEN TO SIGNAL ALL FILES RIPPED!
		ALWAYS LOAD TO \$020000 CALL ORIG LOADER (A0 = CURRENT FILENAME) SCREEN TO BLUE WAIT LMB (SO WE CAN USE AR TO SAVE FILE)
		POINT TO NEXT FILENAME IN TABLE

Illustration 4: Modified bootblock to rip files

Everything is clearly commented above, so lets just look at the process for ripping all the files using our newly-modified copy:

1. Boot from copy, wait for green screen.
2. Swap disk to original, press LMB.
3. Screen goes red while file is loaded.
4. Screen goes blue when loading has finished.
5. Enter AR, "m 7f004" shows current filename ptr (eg: 0298), then "m 0298" to see filename.
6. Save with "sm 1:<filename>,20000 <A0 value>", a0=end address of loaded file, all files begin at \$20000.
7. Exit AR with "g" and press LMB to load next file, back to step 3 until all ripped (screen flashes).

For example:

```
p
D0=00000000 00000600 00000600 00000000 00000000 00000000 FFFFFFFF 00000000
A0=000333A0 00001357 00000438 00FE86EE 00002E20 0007F084 00DFF000 0007E000
PC = 0007F038 USP = 0007DFF8 SR = 2710 T=0 S=1 I=111 X=1 N=0 Z=0 V=0 C=0
m 7f004 ————— HOLDS PTR TO CURRENT FILENAME
:07F004 02 98 45 F9 00 00 04 38 B5 C8 66 0C 33 F9 00 DF ..E....8..f.3..B
m 298 ————— FIND FILENAME FOR SAVING
:000298 61 6E 69 6D 2E 70 72 67 00 00 80 01 65 01 29 4F anim.prg,...e.)0
sm 1:anim.prg,20000 333a0 ————— END ADDRESS IS IN REG A0
Disk ok
g
No known virus in memory!
Ready.
p
D0=00000000 00000600 00000600 00000000 00000000 00000000 FFFFFFFF 00000000
A0=00067F50 000019E9 00000438 00FE86EE 00002E20 0007F084 00DFF000 0007E000
PC = 0007F038 USP = 0007DFF8 SR = 2710 T=0 S=1 I=111 X=1 N=0 Z=0 V=0 C=0
m 7f004
:07F004 02 A8 45 F9 00 00 04 38 B5 C8 66 0C 33 F9 00 DF ..E....8..f.3..B
m 2a8
:0002A8 69 6E 74 72 6F 2E 70 72 67 00 80 4C 8D 04 53 DF intro.prg..L..SB
sm 1:intro.prg,20000 67f50
Disk ok
```

Illustration 5: Example of ripping files

Swap disks before saving file "lev1.prg", and everything will fit onto 2 blank disks (there is 1mb of data total, so it won't fit on a single disk).

Incidentally, if you look more closely at the original loader you will see it features an encoded table of 16 different MFM Sync values - this might have been annoying if you were dealing with a warped original but since we have the physical disk we can completely ignore this!

Now after 15 lines of code and a few minutes of loading/saving we have all the files successfully ripped to our workdisks, we can see that we're going to have to pack at least some files for our crack to be on 1-disk like the original. Thankfully there are no packed files on the original, so this won't really be a problem. I started by using Imploder to pack the 4 files loaded by the first piece of code at \$07E000:

NAME	ORIG SIZE	PACKED
anim.prg	78,752	39,144
load.dat	48,032	N/A
intro.prg	294,736	184,094
alien.prg	48,102	20,424

Packing the loading picture "load.dat" only saved about 3k, so lets ignore that file. A quick calculation of the overall size if we pack the other 3 files gave me a total of about 850k. Since we know an Amiga disk holds 880k, this will be enough, there is no need to pack any other files.

At this point we know the first code loaded by the bootblock to \$07E000 does the following:

1. Loads/execs "anim.prg", "Hi-Tec Premier" animation.
2. Loads/displays "load.dat", loading picture - we need the copperlist + setup code for this in our crack.
3. Loads/execs "intro.prg", intro animation.
4. Loads/execs "alien.prg", the main game exe.

We also know that "alien.prg" probably contains another loader, since there are many other files in the filetable not referenced/used yet and after loading "alien.prg" it is jumped into directly, not called with JSR.

If we look at the loader code at \$07E1CA we can see the first instruction is MOVEM.L D1-D7/A2-A6,-(A7). Reset and let the game load fully then enter AR and search for this instruction:

```
>>> f 48 e7 7f 3e,37820
```

We start searching from \$037820 since this is the base address of "alien.prg", and we only find 1 match at \$03FBFE. This is probably the loader, but to be certain lets look for references to it:

```
>>> fa 3fbfe 37820
```

The first hit is at \$03F4BC, a quick look at the code around this location shows that this is indeed the loader and this time it's called to load "title.dat" to \$01C600.

Now that we have enough information to produce a working crack (you can check out all the references to the loader and reassure yourself that all the files in the filetable are loaded via this loader) it is time to start coding!

I decided just to write my own code to replace the original code loaded to \$07E000, ripping the pieces I needed such as the copperlist + setup code for displaying "load.dat", since we need to decrunch 3 of the files and we are using a different loader/filetable anyway. The code is in the file "alienstart07e000.s" in the attached archive, so I won't repeat everything here, but perhaps I should mention the differences from the original:

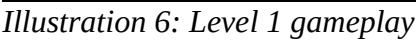
1. Loads/depacks/execs "anim.prg", "Hi-Tec Premier" animation.
2. Loads/displays "load.dat", loading picture
3. Loads/depacks/execs "intro.prg", intro animation.
4. Loads/depacks/patches/execs "alien.prg", the main game exe.

We also check a flag controlled by the crack intro (not supplied, write your own!), if this flag is set to 1 then we will skip steps 1-3 above (all introduction stuff) and go directly to loading the main game exe.

The patching of "alien.prg" is simply a few NOPS (zap the filetable loading and loader init stuff) then overwriting the first instruction of the loader with a JMP MYLOADER before we begin, since after looking around a bit I discovered the initial code at \$07E000 always remains in memory despite never being used once the main game file is executed - we just redirect it to our existing loader, and job done!

Our new filetable requires 12 bytes per entry (original used 16), and is simply the first 8 chars of the filename + a longword which is the disk offset. Since the minimum filelength of the original names was 7, and all names have a terminating null, only storing the first 8 chars of each filename is enough to identify them uniquely. To get the filelength for each load we simply subtract the diskoffset for this file from the diskoffset of the next file - this saves wasting another 4 bytes to store the filelength. As always, we add a little routine which converts the loader arguments used by the game into the arguments our replacement loader expects - this time I've chosen to use a byte-loader by Melon Design (ripped from the game "Naughty Ones") so that we don't waste any unnecessary disk space.

Once this code is assembled and written as binary to "boot07e000.bin", we can turn our attention to creating our crackdisk proper. All the code is contained in "aliencrack01.s", again I won't detail it too much here but basically all it does is load our new code to \$07E000, and copy our filetable to \$200 before any loading occurs (since there is no "load filetable" code anymore).



The Amiga version was a simple crack, but what about the Atari ST version?

Atari ST cracking

"Alien World"

(c) 1992 Hi-Tec

Things you will need:

1. The original of "Alien World" (c) 1992 Hi-Tec. The PASTI image from www.atarimania.com is the one I've used here.
2. An Atari ST or a copy of STEEM.
3. Devpac, or your personal favourite assembler, from www.dhs.nu (in the files/coding-tools section).
4. Bugaboo debugger, from www.dhs.nu (in the files/coding-tools section). *Note that this is part of the Turbo Assembler package!*
5. Easyrider disassembler, from www.dhs.nu (in the files/coding-tools section).
6. Speedpacker v3 (Google it!)
7. A copy of "Atari ST Internals" to explain all things ST (since we're only familiar with Amiga hardware!).

Before we begin, I should point out that I'm not really the person to teach anyone anything about Atari ST cracking! I just thought it might be interesting to crack the ST version of a game we previously cracked on Amiga to see what the differences (and indeed the similarities) were between the two platforms.

One difference you will spot immediately is that the ST does not have any equivalent to the Action Replay cartridge on Amiga - in fact there is no cart which has a "magic button" (level 7 irq/NMI) for jumping into code at a time of your choosing. The (very poor) carts available are only capable of examining memory after a soft reset, and are of fairly limited use. Something else to note is that almost all the tools available are inferior to their Amiga cousins (the assemblers don't let you create a diskimage in ram and write it out directly to disk, the disassemblers make a poor job of producing working code) so just keep that in mind as we take our first steps into the wonderful world of ST cracking!

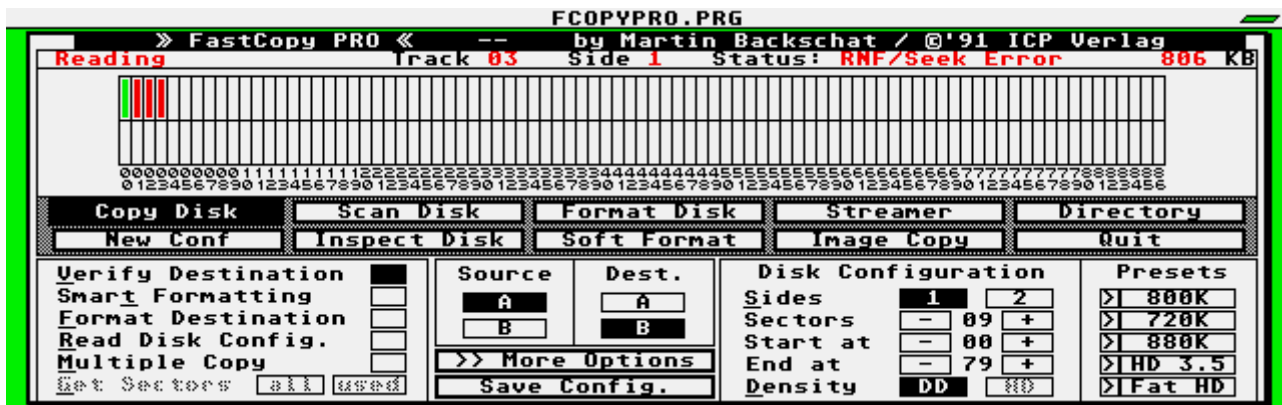


Illustration 7: FcopyPro Alien World ST

As you can see from the obligatory picture of an attempt to copy the original the disk is not in a copyable format, so our next step is to look at the bootblock using the EasyRider 4 disassembler (referred to as ER4 from this point on). If you load ER4 from harddisk as I do, it will have "C:" as the active drive so don't attempt to disassemble the bootsector yet. First we need to go to the File menu and choose "Load program" then change the drive to "A:". It will then take a directory of the disk (finding nothing since it's a custom format) and you can exit from there then use the "Bootsector" option to view the floppy's bootsector:-

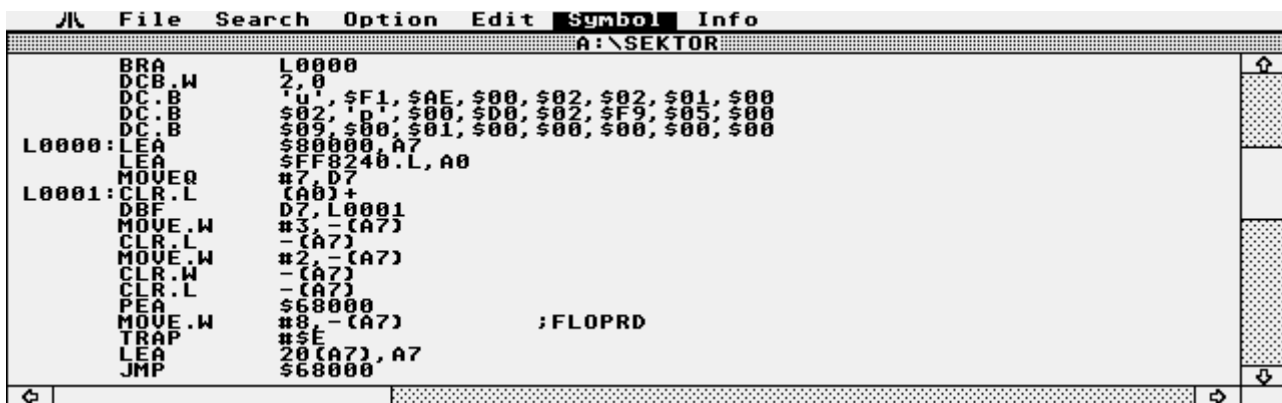


Illustration 8: ER4 bootblock disassembly ST

It sets the stack to \$80000, clears all colours to 0 and then comes part we are interested in:-

```

MOVE.W    #3, -(A7)                ;num of sectors
CLR.L     -(A7)                    ;disk side & track num
;^^ (both args are .w so this sets both to 0) ^^
MOVE.W    #2, -(A7)                ;start sector
CLR.L     -(A7)                    ;device (0 is drive A)
CLR.L     -(A7)                    ;reserved, always 0
PEA       $68000                   ;load address
MOVE.W    #8, -(A7)                ;Floprd()

```

```

TRAP      #14                ;call XBIOS function
LEA        20(A7),A7          ;restore stack to normal
JMP        $00068000          ;jump to loaded code

```

So as we can clearly see it simply reads 3 sectors (512*3=1536 bytes) starting at track 0 sector 2, directly after the bootblock, to \$68000 and jumps to this address.

Incidentally, this might be a good time to explain a little about the layout of a normal ST disk.

As we all know, the Amiga can store 880k on a disk, which is 11 sectors-per-track * 80 tracks * 2 sides = (11*512)*(80*2) = 901120 bytes /1024 = 880k.

The ST (usually) only stores 9-sectors per track = (9*512)*(80*2) = 737280 bytes /1024 = 720k.

Most STs could actually read a 10 sector/800k format disk without problems, and indeed a large number of cracked games used this format. Very few could reliably read an 11 sector/880k format, and as such this was very rarely utilised...but for maximum compatibility it would be nice if we could fit our final cracked version onto a normal 720k ST disk.

We can read this data using ER4 using the "Load sector" command, however if you try to load from sector 2 you will discover that you are sitting halfway through some code and not, as you would expect, at some initialisation code. For some reason we only get the correct code if we tell ER4 to begin reading from sector 1 like so:-

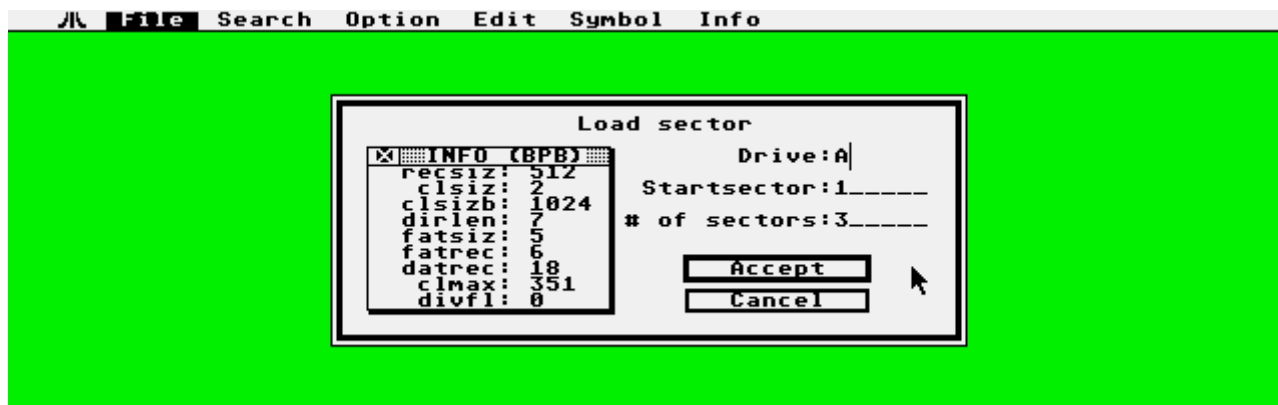


Illustration 9: ER4 load sectors ST

This looks good! We can clearly see from the first few instructions that this is a loader:-

```

L0000:LEA      L0000(PC),A7
        LEA      L003A(PC),A5
        ST        $43E.L
;^^ FLOCK system variable (signal floppy is in use) ^^
        MOVE.B    #$13,$FFFC02.L
;^^ writes to keyboard ACIA, no idea why?! ^^

```

```

MOVE      #$2700,SR      ;supervisor
BSR       L0005           ;init disk, load filetable to $000200
MOVE.L    #$3FFE4,D0
LEA       L003D(PC),A0    ;points to "talk.prg",0 filename
BSR       L0001           ;load "talk.prg" to $03FFE4
JMP       $40000

;^^ jump to loaded code (jmp over .PRG header) ^^
L0001:MOVE.M    A2-A6/D1-D7,-(A7)
        LEA      $FF8604.L,A6

;^^ disk DMA regs accessed as xx(A6) from here on... ^^

```

The word at \$00043E is used by the OS to signal the floppy is in use (named "FLOCK"), so as soon as we see this we can be fairly sure it's about to do something disk-related :) Of course if you are not using the OS for anything (eg: a game or demo) then there is no requirement to do this, but at this moment they haven't taken over any of the interrupt vectors or nuked the system from low-memory so maybe this is why they did so. If we scroll down and check out what "L003D" points to we see the null-terminated string "talk.prg", and because we see a JMP \$040000 instruction immediately after the BSR L0001 we know that this must be the loader proper since there is nothing in memory at \$040000 at this stage in the process - which means that the MOVE.L #\$03FFE4,D0 must be setting the load address... incidentally the file is loaded to \$040000-\$1C because for some reason it includes a real ST .PRG header (ST executable) which is simply jumped over!

This is a perfect time to save the disassembly since we know the loaded code only consists of the loader and a few lines of code to call it to load 1 file ("talk.prg") and jump into it! Since there is no extraneous code for other tasks like showing pictures/other misc crap, this makes ripping a working loader very easy... however, remember what I said about the ST tools being quite poor?! Well if you scroll down a few pages in the disassembly you will see that it has failed to disassemble a small chunk of code (after the 2 instructions BSR L0016, BSR L0025). ER4 has also completely screwed up the amount of buffer space allocated to 3 variables at the end of the loader code, immediately prior to the "talk.prg" string - so before we attempt to write a disk-ripper using the loader code we just borrowed, we need to fix these problems.

Thankfully Bugaboo has a "read sector" command so we can use this debugger to examine the code and manually fix the things ER4 screwed up for us:-

The screenshot shows the Bugaboo debugger interface. At the top, there's a header with columns: Trace, Do PC, Tracrts, Traps, Skip PC, Source, Hexdump, Disasm, List, Switch. Below this, the PC is set to 00018AAC, USP to 000F7FF8, and SSP to 000DC0D8. The disassembly shows instructions like 'ori.b #0,D0' and 'ori.b #0,D1'. Below the disassembly, there's a message: 'X-Soft's Bugaboo 01.7.14 von Markus Fritze und Soren Hellwig'. Further down, it says 'Ein 68000 Prozessor ist aktiv.' and '>Falcon030-Improvements by Chris/AURA and The Innovator/NEWline<'. At the bottom, there are user commands: '\$00018AAC>readsector 0,2,0,68000', '\$00068000>readsector 0,3,0,68200', '\$00068200>readsector 0,4,0,68400', and '\$00068400>d 68000'.

Illustration 10: Bugaboo read sectors ST

(NOTE: I only typed the "readsector" commands separately so you could see how it is used: track,sector,diskside,address - you would simply use the up-arrow and edit the previous line if you were actually doing this!)

The first thing to fix is the section of code ER4 failed to disassemble at all, we know it comes shortly after a MOVE.W D0,\$26(A5) instruction (which is unique in the disassembly we just saved out, so this is a good choice for our search)... after repeated use of the "D" command to disassemble from \$68000 onwards, we find the garbled bytes in our original disasm began immediately after the 2 BSR's and now we can see the code which ER4 failed at:-

```
MOVE.W 38(A5),D0
MULU #$400,D0
LEA $600,A1
ADDA.L D0,A1
MOVE.W (A1)+,36(A5)
MOVEA.L 0(A5),A0
BTST #$FF,10(A0)
BNE L002EFIX
```

Because of the failed disassembly, ER4 had used the label L002E so I simply added another to point to the address of the BNE instruction you see in the snippet above. Now that we have the complete code, it's time to repair the absolutely awful job ER4 made of allocating buffer space for the 3 labels near the end of the loader disassembly, L003A to L003C inclusive.

I should note that I'm not psychic & didn't know that there was a problem here until I attempted to use my ripped loader - when I debugged it I saw that the filename string ("talk.prg",0) was being overwritten so this is why I know the buffer space wasn't correctly allocated (since the filename was in memory directly after the 3 buffer labels mentioned).

The second line of code references L003A so we can easily see this points to \$0684C2 when the code is in memory ("D 68000" and see that the second line is LEA \$0684C2). L003B is referenced 2 instructions after the MOVEM.L at the beginning of the init/load-filetable code at label L0001 so thanks to Bugaboo we can see this points to \$0684C6. L003C is the final label we have to fix, and we can see this is referenced just a few lines after the CMPI.L "DOS1", (A0) instruction... 4 pages later we find it now points to \$0684FC. If we check the position of the "talk.prg" string (which we know is referenced in the first few instructions) we find it at \$06850C... so now we know all the information required to fix our ripped loader code.

L003A = \$0684C2

L003B = \$0684C6, so we allocate \$04 bytes space @ L003A
(\$0684C6 - \$0684C2)

L003C = \$0684FC, so we allocate \$36 bytes space @ L003B
(\$0684FC - \$0684C6)

L003D = \$06850C, so we allocate \$10 bytes space @ L003C
(\$06850C - \$0684FC)

Now that we have source to a functioning loader, we can get on with the process of ripping the original files. If you check the source to "AWRIPPT1.S" (or part 2) you can see I've just added a little piece of code to loop through the filetable and load each file from the original disk in drive A: and save to drive B: - the reason for 2 ripping programs is simply each one handles ripping half the disk, since the original files are too big to fit on a single disk. This is done using the GEMDOS functions FCREATE/FWRITE/FCLOSE - on the ST these routines are called via a function number being pushed onto the stack before a TRAP #1 instruction (and in a similar manner other functions known as XBIOS routines are called via TRAP #14, as you saw in the bootblock).

I had some problems with random crashes/infinite saving (saving a file that never stopped, until the disk was full) initially, until I made 2 changes to my ripping code - calling the "disk init" code before every file seemed to save more files correctly before crashing, but it still failed so I simply added a useless time-wasting routine that flashes the screen for a while after every file and that seemed to fix everything! Certainly easier than properly debugging the code :)

Also, I initially used the filelength from the game filetable to save out my ripped files...however when I disassembled "ALIEN.PRG" to look for further loaders, I found that a portion of the code was missing. The file was loaded to \$16800 and the first instruction is JMP \$243AE, but if we used the filelength from the filetable of \$D327, there would obviously be no code here ($\$16800 + \$D327 = \$23B27$) so instead I used the value in A0 after calling the loader and subtracted the start address from this to calculate the size of each file before saving - for "ALIEN.PRG" this now gave me a filelength of \$F1D2, and disassembly showed code @ \$243AE onwards which suggested it was ripped correctly this time!

Also worth noting here is the file "end.dta" fails to rip correctly, but not because of anything we've done - the original PASTI image doesn't include data beyond track 79, so it is missing the end of the file! Thankfully, it looks like this file is identical on both the Amiga and ST versions, so this problem was fixed by copying the file we ripped during the Amiga crack and using it instead.

It's time to move onto the important part of any crack - getting everything together and working! :) We know there are 2 loaders (potentially more at this point, but certainly a minimum of 2) that we have to replace with our file loader (ripped from "Chaos Engine" by Dr.D/TCD, I think I downloaded it from Atari Forum many years ago). The game loader that we ripped & used to extract all the files was originally meant to load/execute "talk.prg" so the next step is to disassemble this and find the loader code. Just as before, load ER4 and select "talk.bin" ("talk.prg" originally but we renamed all ".prg" files when we ripped them), and after a little stub which relocates everything to \$07E000 we can soon spot what we're looking for:-

```
        LEA        $FF8240.L,A0          ;palette ram
        MOVEQ      #$F,D7
L0002: CLR.W       (A0)+
        DBF        D7,L0002              ;clear all cols to black
        BSR        L000E                  ;installs VBL irq
;^^ (which simply does vblcount++ every frame) ^^
        BSR        L001A                  ;disk init
        LEA        L0012(PC),A0          ;"anim.prg"
        MOVE.L     #$20000,D0            ;load addr
        BSR        L0016                  ;call loader
        BSR        L000C                  ;wait vbl
        JSR        $20000                 ;JSR loaded code
        BSR        L000E                  ;install VBL again
;^^ (nuked by "anim.prg") ^^
        MOVE.L     #$10000,D0            ;screen display address
        LEA        $FF8201.L,A0
        LSR.W      #8,D0
        MOVE.B     D0,2(A0)              ;video base medium
        SWAP       D0
        MOVE.B     D0,(A0)               ;video base high
        LEA        L0014(PC),A0          ;"load.dat"
        MOVE.L     #$10000,D0            ;load addr
;^^ (which is set to be used for screen display, see above) ^^
        BSR        L0016                  ;call loader
```

We can see the usual arguments to the loader (A0=ptr to filename, D0=load addr) so it's obvious that L0016 is the loader routine, but if we scroll down to check it manually we also see the filenames for the 4 files loaded by "talk.prg": "anim.prg", "load.dat", "intro.prg" + "alien.prg" immediately above the first line of loader code. Time for a short pause in the disassembling to do some packing!

Examining the ripped files, we can see that the total size of all files exceeds the amount of data we can fit on 1 disk - a total of 898,907 bytes. We know that we have to fit in under 737,280 bytes to fit on a normal 9 sector/track format (actually quite a bit less than that if we use the OS filesystem rather than writing raw data to disk, since we need space for the dir structure/filenames etc.), so it's clear that we have to pack some files.

Just as certain packers became popular on the Amiga scene, the ST had it's own favourites... Automation packer, Speedpacker v3, Atomik and the (in)famous "Ice!" packer (JAMPack) were probably used in more than 90% of cracks on Atari. So to keep it as traditional as possible I chose to use Speedpacker v3 by the FireHawks (more commonly known as SP3) for this crack.

There are 4 files loaded by "TALK.PRG", 3 of which are obvious candidates for packing, namely "ALIEN.BIN", "ANIM.BIN" and "INTRO.BIN". The other file "LOAD.DAT" is the loading screen, and presumably because there are no areas of blank/repeated gfx it does not compress well so we'll skip this one. We should also pack the 2 largest files remaining, "ALIEN.MOD" and "END.DAT", because otherwise we'll run out of space if at least 1 of these 2 isn't compressed! SP3 has an option to pack all the files on one disk and write them to another, so the simplest way to proceed is to copy these 5 files to a blank disk in "A:", and insert another blank disk into "B:" to save the packed files.

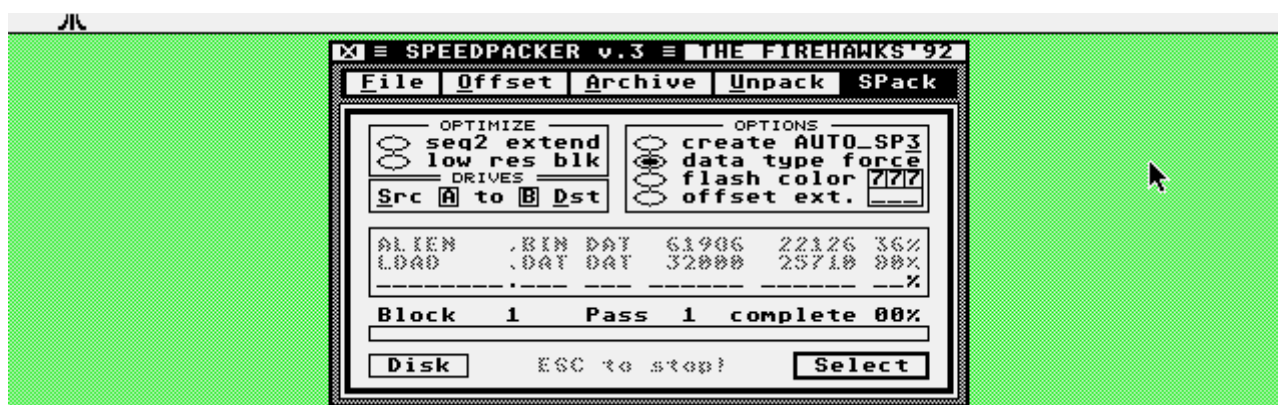


Illustration 11: SP3 packing files ST

One thing to note here is that the files are saved with the same filenames as the input files, so the first thing you should do is go to drive B and rename the files (File->Show Information) to end in ".SP3" instead of ".BIN" or ".DAT". Time to take a quick look at how the packer performed:-

Filename	Orig size	Packed size	Saved bytes
"ALIEN.BIN"	61906	22126	39780
"ANIM.BIN"	72256	34036	38220
"INTRO.BIN"	237760	127870	109890
"ALIEN.MOD"	106916	69168	37748
;^^ (NOTE: renamed to "AWMOD.SP3") ^^			
"END.DAT"	97971	70926	27045

Original total length = 898907

Total saved bytes = 252683

New total length = 898907-252683 = 646,224 bytes!

This will certainly fit on a normal ST floppy so now we only need to replace "talk.prg" with our own code (fileloader + SP3 depack), and patch the loader/depack into "alien.prg" before executing the main game. I went back to ER4 and the disassembly of "talk.prg" to see which pieces of code I needed to rip, only to find it had again buggered up parts of the code:-

L000E:

<...uninteresting code removed...>

```

        LEA        L000F(PC),A0          ;pointer to VBL handler code
        MOVE.L     A0,$70.W             ;VBL irq vector
        MOVE       #$2300,SR
        RTS

```

L000F: DC.B blah,blah,blah

Well since we can clearly see L000F points to code, we need to fix this. In exactly the same way as we fixed the earlier code, we just load "talk.bin" into Bugaboo and disassemble it, looking for the MOVE.L A0,\$70.W instruction above and fix the missing code by hand... fortunately it's a tiny and simple routine:-

L000F:

```

        move.l     a0,-(a7)
        lea        vblcount(pc),a0
        addq.w     #1,(a0)
        move.l     (a7)+,a0
        rte

```

vblcount: dc.w 0

There was another piece of code which apparently converts an Amiga palette to ST colours for the title screen - since I couldn't see any purpose in doing this realtime (rather than storing the correct palette as DC.W col1,etc) I actually executed this code on my Amiga to get the correct palette as fast as possible :P You could of course achieve the same thing on ST with Devpac or Bugaboo and write it to a file, or even just include the original horrible code from "talk.bin", this was just lazyness on my part...

At this stage, I coded my replacement for "talk.prg" with the file-loader mentioned previously, and an optimised SP3 depacker by Mr.Ni/TOS-Crew (also posted to Atari Forum, thanks again). Since we already have a disk containing the files we compressed, I just copied it to that and used this as my work disk - obviously replacing the final JMP \$016800 which would have started the game (and crashed when it tried to do further loading with the original game routines) with a flashing screen so I could see things had worked up to this point...success! Now we can copy the remaining files that didn't require packing to our workdisk before the next step.

The next thing to do is disassemble "ALIEN.BIN" ("alien.prg" originally) using ER4 in order to locate the next loader. This time we will save out the sourcecode as a disassembly with memlocations & hexbytes of all the opcodes as "ALIENDIS.S".

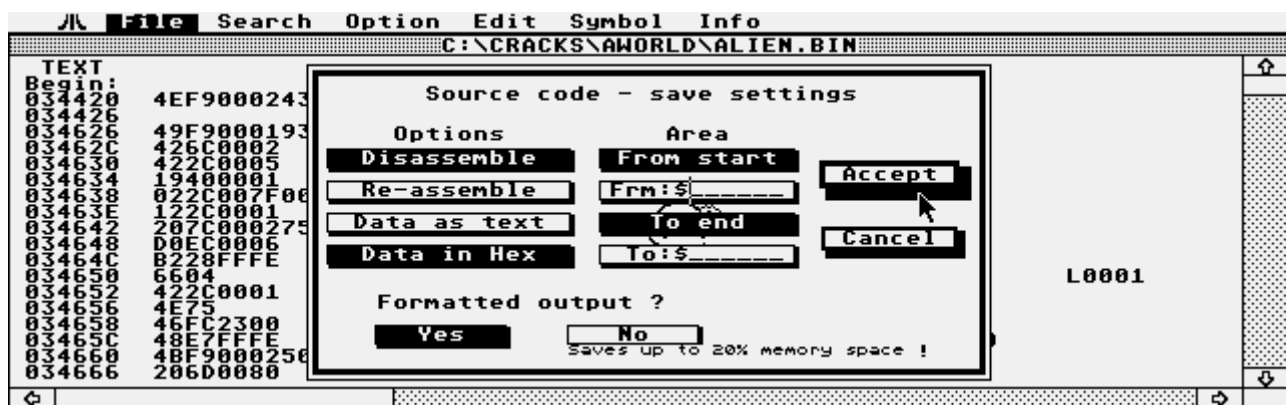


Illustration 12: ER4 Alien full disassembly ST

Why did we save it out as a full disassembly? The first instruction is JMP \$243AE, and since we can't set an absolute address for the disassembly we need to work out which label in the disasm this would point to:-

$\$243AE \text{ (jmp addr)} - \$16800 \text{ (start addr)} = \$DBAE$

Our disassembly begins at \$34420, so $\$34420 + \$DBAE = \$41FCE$ is the address we need to search for in "ALIENDIS.S". We find this leads to 4 instructions before L037C in our disasm, but the important thing to find is the loader.

We can find this easily by looking for the MOVEM.L A2-A6/D1-D7, -(A7) instruction that began the loader we already ripped - and sure enough we find only one occurrence of this opcode at L032C.

Now that we know our loader is labelled L032C, we need to find a reference to it so we can track down the filename pointers - the first occurrence passes L0188 as the filename, however if we look at what this points to we see something odd. Prior to the filename, there is a \$01 byte (DC.B \$01,"title.dat",0) and if we check out the other filenames (either search for other calls to the loader, or simply look around L0188 since all the filenames are together in memory) we discover they all begin with either a \$00 or a \$01 byte preceding the filename proper. Since our loader will fail miserably if we pass it these filename pointers, we just need to remember to add 1 to the pointer in our loader 'stub' (the code that converts game args to our loaders args and calls decruncher, etc).

We must also overwrite the filenames in memory with our new filenames, since some files are now packed and we changed ".PRG" to ".BIN" etc... so we have to search through the disassembly and note down the offsets of each filename pointer and the loader itself, so we can patch everything to our liking.

I won't bore you by listing it all here, you can find the relevant details in the final crack sourcecode included with this tutorial ("AWCRACK.S").

Since I had already cracked the Amiga version, I suspected that the loader used in my "talk.prg" replacement was still in memory (above \$07E000 somewhere), so the easiest way to patch the ingame loader within "alien.prg" was just to copy our little stub directly over the loader code before jumping to the start address:-

```
loaderstub:
    move.l    a1,-(a7)
    move.l    d0,a1
;^^ convert from D0=load address (GAME) to
;A1=load address (FILELOADER) ^^
    addq.l    #1,a0
;^^ game filename ptrs point to a 0 or 1 byte BEFORE the string
;so we skip this ^^
    move.l    a1,-(a7)
;^^ save load address for decruncher ^^
    jsr       $7e000+(FILELOAD-reloc_start)
    move.l    (a7)+,a0
;^^ decruncher wants pointer in A0 ^^
    jsr       $7e000+(SP3DEPACK-reloc_start)
;^^ always call depacker, it will just return if file is not
;packed anyway... ^^
    move.l    (a7)+,a1
    rts
loaderstub_end:
```

Great, I thought, we have our loader/decruncher hooked into the game, we ripped all the files & packed some where necessary, now we just need to test our cracked game...but it appears there might be a problem!

Everything works up to the main menu, credits begin & music plays, but why won't the game start when I press fire?! I'm sure it used to!

Honestly, this baffled me for some time... I thought the loader might be messing up the ACIA chip which handles keyboard/joystick input, but after speaking to GGN/D-BUG (thanks for your patience) I ruled this out by inserting some code which flushed/reset the ACIA on loader exit, which completely failed to fix the problem! The exception vector @ \$118 points to the handler for keyboard/joystick stuff, so let's see what it points to.

Searching through our disassembly for \$118 found this snippet of code which is clearly setting up various irq handlers:-

```
03D9D2  61000134          L018E:BSR          L0196
03D9D6  46FC2700          MOVE             #$2700,SR
03D9DA  13FC000000FF8260  MOVE.B          #0,$FF8260.L
03D9E2  203C0002016E      MOVE.L          #$2016E,D0
03D9E8  41F80064          LEA              $64.W,A0      ;level 1 irq
03D9EC  20C0              MOVE.L          D0,(A0)+
03D9EE  20C0              MOVE.L          D0,(A0)+
03D9F0  20C0              MOVE.L          D0,(A0)+
03D9F2  20FC000200E0      MOVE.L          #$200E0,(A0)+
03D9F8  20C0              MOVE.L          D0,(A0)+
03D9FA  20C0              MOVE.L          D0,(A0)+
03D9FC  20C0              MOVE.L          D0,(A0)+
03D9FE  23FC0002016E00000118  MOVE.L          #$2016E,$118.L
      ;^^ ACIA irq ^^
03DA08  23FC0002018000000120  MOVE.L          #$20180,$120.L
      ;^^ Timer B (HBL counter) irq ^^
03DA12  23FC0002017000000134  MOVE.L          #$20170,$134.L
      ;^^ Timer A irq ^^
```

After working out what \$2016E equates to in my disassembly (\$3DD8E), I think I can see why the game doesn't start... it points to an RTE instruction! This means there is simply no handling of joystick input, the handler simply returns... obviously this wasn't the case with the original game (or, non-game, since you wouldn't be able to play it) - I smell the foul odour of copy-protection!

Checking the rest of the disassembly for anything writing to \$118 didn't produce anything useful, so since the code above which sets up all the various irqs (and is responsible for pointing the \$118 handler to an RTE) is labelled L018E lets find references to this routine - there is only one, and as it turns out it's the very first subroutine called. Remember the

JMP \$243AE pointed to \$41FCE in my disassembly:-

```
041FCE  4FF900016A06      LEA          $16A06,A7
041FD4  4BFA0C7C           LEA          3196(PC),A5      ;L03E5
041FD8  6100B9F8           BSR          -17928(PC)      ;L018E
041FDC  610002CE           BSR          718(PC)         ;L0391
```

Let's take a look at what happens in the next subroutine, L0391:-

```
0422AC  46FC2700           L0391:MOVE      #$2700,SR
0422B0  41F900000200       LEA          $200.L,A0
;^^ I thought filetable base, but used to offset to $118 below ^^
0422B6  203C3D3D7903       MOVE.L       #$3D3D7903,D0      ;key1
0422BC  222D00A4           MOVE.L       164(A5),D1
;^^ $A4(A5), A5=$25032 so $250D6 ^^
0422C0  B380              EOR.L        D1,D0
;^^ xor unknown long from $250D6 with key1 ^^
0422C2  3200              MOVE.W       D0,D1
0422C4  E249              LSR.W        #1,D1            ;result / 2
0422C6  6500000C           BCS          12(PC)          ;L0392
0422CA  0680000243AE       ADDI.L       #$243AE,D0
;^^ $243AE is the JMP address to where code begins ^^
0422D0  2140FF18           MOVE.L       D0,-232(A0)
;^^ -$E8(A0), so if A0=$200 this writes to $118! ^^
0422D4  46FC2300           L0392:MOVE      #$2300,SR
0422D8  223900000090       MOVE.L       $90.L,D1
0422DE  B2BC00080001       CMP.L        #$80001,D1
0422E4  6500000E           BCS          14(PC)          ;L0393
0422E8  203C00080000       MOVE.L       #$80000,D0
0422EE  9280              SUB.L        D0,D1
0422F0  6100F4F0           BSR          -2832(PC)      ;L032A
0422F4  6100F5A8           L0393:BSR      -2648(PC)      ;L0330
0422F8  6100003A           BSR          58(PC)         ;L0394
0422FC  203C00001A00       MOVE.L       #$1A00,D0
042302  41FAB28D           LEA          -19827(PC),A0    ;L0187
;^^ points to (DC.B 0,"ship.dat",0) ^^
```

```

042306 6100F510          BSR      -2800(PC)      ;L032C
;^^ (loader) ^^
04230A 41F900001A00      LEA      $1A00.L,A0
042310 43F90004E7F0      LEA      $4E7F0,A1
042316 45FAA75A          LEA      -22694(PC),A2      ;L0183
04231A 2B7C00014C000098  MOVE.L  #$14C00,152(A5)
042322 6100D530          BSR      -10960(PC)      ;L020A
042326 2B6D0098009C      MOVE.L  152(A5),156(A5)
04232C 23C90000008C      MOVE.L  A1,$8C.L
042332 4E75              RTS

```

Well that would explain why I didn't find any references to \$118 that looked like protection - the routine accesses it via an offset from \$200! But we can clearly see we're in the right place from the calculations before the overwrite of the pointer at \$118... sadly there isn't an obvious/hard-coded value used in the code, it is fetched from \$250D6. A quick search for this, and there is only 1 occurrence:-

```

041904 41F9000250D6      LEA      $250D6,A0
;^^ the long we are interested in ^^
04190A 43F900000600      LEA      $600.L,A1      ;diskbuff
041910 3E3C03FF          MOVE.W  #$3FF,D7
041914 0C1900A1          L0334:CMPI.B  #$A1,(A1)+
041918 6700000A          BEQ      10(PC)      ;L0335
04191C 51CFFFFFF6          DBF      D7,-10(PC)      ;L0334
041920 6000000E          BRA      14(PC)      ;L0336
041924 43E9001A          L0335:LEA    26(A1),A1      ;$1A(A1)
041928 10D9              MOVE.B  (A1)+,(A0)+
04192A 10D9              MOVE.B  (A1)+,(A0)+
04192C 10D9              MOVE.B  (A1)+,(A0)+
04192E 10D9              MOVE.B  (A1)+,(A0)+
;^^ copy 4 bytes from somewhere in diskbuff to $250D6 ^^
<...>

```

So the value in \$250D6 is copied there by this routine from somewhere in the diskbuff...the code scans the buffer for the value \$A1, and when it's found it copies 4 bytes from this position+\$1B (remember CMPI.B #\$A1,(A1)+ so the buffer pointer is already incremented before the LEA \$1A(A1),A1 is executed above. It looks like this value is in the diskbuffer after the filetable is loaded - I quickly hacked together a disgusting piece of code ("RIPKEY01.S") which patches "alien.prg" before execution so that after the key is copied to \$250D6 we "borrow" the key and copy it over the filetable (\$200+) then freeze... upon reset (soft reset, NOT hard) I simply loaded Bugaboo and checked memory:-

Illustration 13: Bugaboo find protection key ST

It would appear the magic key is \$C2C2C2C3! Since we know this key is used to produce a valid vector address for the \$118 handler (ACIA, keyboard/joystick) we should check if this key would produce a pointer to something useful.

Back to the pointer-calc code, this time with the values we know now:-

```
0422B6  203C3D3D7903          MOVE.L    #$3D3D7903,D0
;^^ D0 = $3D3D7903 ^^

0422BC  222D00A4          MOVE.L    164(A5),D1
;^^ D1 = $C2C2C2C3 ^^

0422C0  B380              EOR.L     D1,D0
;^^ D0 = $FFFFBBC0 ^^

0422C2  3200              MOVE.W    D0,D1
;^^ D1 = $FFFFBBC0 ^^

0422C4  E249              LSR.W     #1,D1
;^^ D1 = $7FFFDDE0 ^^

0422C6  6500000C          BCS       12(PC)
;^^ branch not taken because we have the correct key :) ^^

0422CA  0680000243AE        ADDI.L    #$243AE,D0
;^^ $FFFFBBC0 + $243AE = $0001FF6E is the real $118 handler ^^
```

The only thing left to do is add the code to patch "alien.prg" before we jump into it. If you want all the details please check the attached source code ("AWCRACK.S"), but I'll summarise it here:-

1. We patch all the filenames (using a table of offsets and a 2nd table of our replacement names) since there is no filetable now - remember we changed lots of filenames, and 2 files are packed now ("mod.alien" is now "AWMOD.SP3", "end.dta" is now "END.SP3").
2. Insert an RTS to immediately return from a couple of disk-init/filetable-loading routines.
3. Copy our 'stub' over the game loader, which fixes the arguments to the format our file-loader expects, and calls the SP3 depacker after each load.
4. Patch the protection check - just overwrite the code which searched the diskbuff and copied the key to \$250D6 with a routine that simply moves \$C2C2C2C3 to \$250D6 so that any routines which rely on this value should work correctly.

We assemble our crack, copy the resulting program to our workdisk, and cross our fingers!



Illustration 14: Level 1 gameplay ST

Conclusions

So what have we learned from all this hard work cracking the Amiga and Atari ST versions of the same game?

There are several similarities in the actual game code (to be expected), eg: both games store the filetable at \$000200 and use \$00600 as the diskbuffer. The Amiga version was obviously the 'lead' platform, judging by the palette conversion code in the title pic display routines – indeed it's a better looking game on Amiga since the ST version has a plain/repeating grey background instead of the more atmospheric/detailed red mountainscape in the Amiga version (presumably due to screen-clearing code it was necessary to use a repeating pattern on ST instead).

In terms of cracking, the ST version was a lot more work for 3 main reasons:-

1. The tools on ST are vastly inferior to those available on Amiga.
2. There was extra protection on the ST version (the joystick-reading stuff).
3. I'm a lot more experienced with Amiga cracking :)

Incidentally, this tutorial was delayed due to a bug I couldn't understand with files being corrupted in RAM on the ST. After re-ripping all the files and painstakingly retracing my steps, it turned out to be the file-loader that I was using which randomly corrupted memory after the end of the loaded file!

If I was going to do any more Atari ST cracking (I'm not!), I would definitely use a different disassembler (there is one called "Desert Drain" which looks promising). Easyrider 4 wasted so much of my time having to go back and fix poorly or completely incorrectly disassembled code that I would never use it again.

I hope you enjoyed reading this tutorial more than I "enjoyed" writing it!

Thanks to Musashi9 for playtesting both versions, and greetings to all the regular Flashtro visitors for keeping the scene nostalgia fresh.

-WK, March 2016