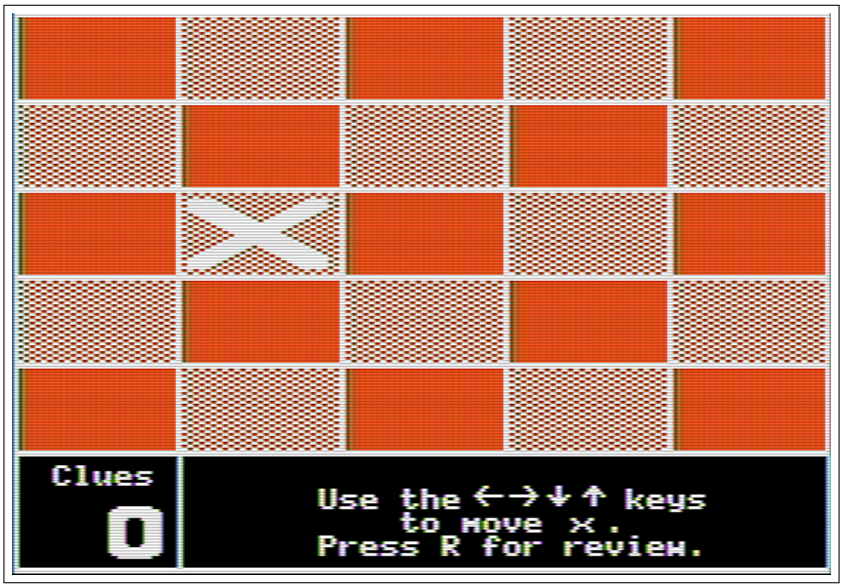
Safari Search



2014-11-11

Safari Search

A 4am crack 2014-11-11

"Safari Search: Problem Solving and Inference" is a 1985 educational game distributed by Sunburst Communications. It was designed by Thomas C. O'Brien, with graphics and programming by Jim Thomas.

[The copy protection is similar to "Exploring Science: Temperature," also distributed by Sunburst. This write-up is therefore quite similar to that one, with a few corrections and the whole thing about the disk volume number.]

COPYA fails miserably and immediately. EDD 4 bit copy gives read errors on tracks $13 and up. Oddly, when I look at the original disk with the Copy][+ nibble editor, those "unreadable" tracks do appear to be formatted and to contain data. Needless to say, the copy does not work.

Further inspection in the nibble editor reveals that each track has a different address and data prologue:

Track	Address	Data
$00	D5 AA 96	D5 AA AD (normal)
$01	D5 AA 97	D5 AA AE
$02	D5 AA 9A	D5 AA AF
$03	D5 AA 9B	D5 AA B2
$04	D5 AA 9D	D5 AA B3

And so on.

```
Time for boot tracing with AUTOTRACE.

[S6,D1=original disk]
[S5,D1=my work disk]

]PR#5
...
CAPTURING BOOT0
...reboots slot 6...
...reboots slot 5...
SAVING BOOT0
CAPTURING BOOT1
...reboots slot 6...
...reboots slot 5...
SAVING BOOT1
SAVING RWTS

For those of you just tuning in, my
work disk runs a program I call
"AUTOTRACE" to automate the process of
boot tracing. For disks that use an
entirely custom boot process, AUTOTRACE
just captures track 0, sector 0 (saved
in a file called "BOOT0") and stops.
For "DOS 3.3-shaped" disks, which load
in the more-or-less the same way as an
unprotected DOS 3.3 disk loads, it can
also capture the next stage of the boot
process (to a file called "BOOT1").
"BOOT1" is usually sectors 0-9 on track
0, which are usually loaded into memory
at $B600..$BFFF. (Of course, there are
exceptions to every rule.)
```

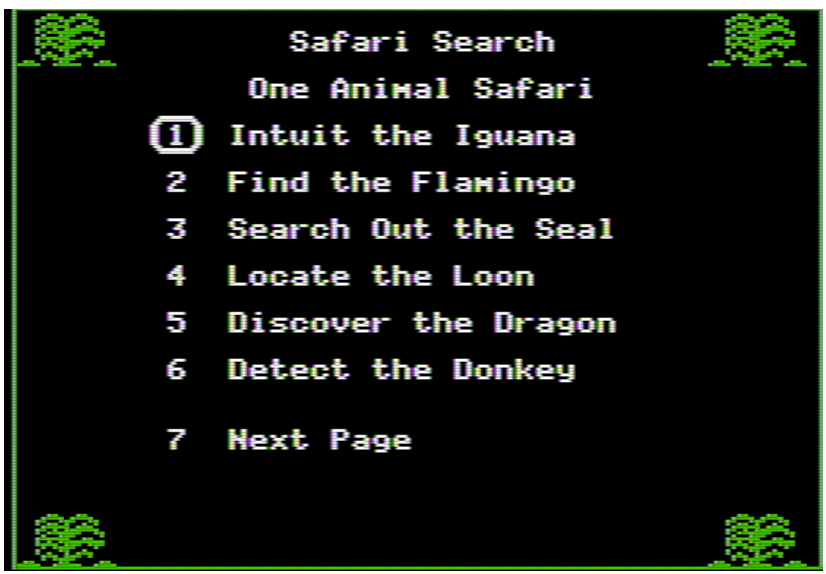
If the boot1 code is "DOS 3.3-shaped," there's a good chance I'll be able to use a tool called Advanced Demuffin to convert the disk from whatever weird format it uses to store its data into a standard disk readable by unprotected DOS 3.3. In this case, the RWTS was close enough to normal for my AUTOTRACE program (which spot-checks a few locations in memory to guess at its "normalcy"), so it extracted the RWTS routines from $B800..$BFFF and saved them into a third file called "RWTS".

If anything looks fishy or non-standard, AUTOTRACE just stops, and I have to check the files it saved so far to determine why. But in this case, it ran all the way through, automatically capturing BOOT0, BOOT1, and RWTS files. Now I can use Advanced Demuffin to convert the disk to a standard format.

```
]S6,D1=original disk]
]S6,D2=blank disk]
]S5,D1=my work disk]

]PR#5
...
```

```
]BRUN ADVANCED DEMUFFIN 1.5

[press "5" to switch to slot 5]

[press "R" to load a new RWTS module]
  --> At $B8, load "RWTS" from drive 1

[press "6" to switch to slot 6]

[press "C" to convert disk]
```

This disk is 16 sectors, and the default options (copy the entire disk, all tracks, all sectors) don't need to be changed unless something goes horribly wrong.



```
--v--

TRK:................R.................
+.5:
    0123456789ABCDEF0123456789ABCDEF012
SC0:................................
SC1:................................
SC2:................................
SC3:................................
SC4:................................
SC5:................................
SC6:................................
SC7:................................
SC8:................................
SC9:................................
SCA:................................
SCB:................................
SCC:................................
SCD:................................
SCE:................................
SCF:................R.................
====================================
16SC $00,$00-$22,$0F BY1.0 S6,D1->S6,D2

--^--
```

Grr. Track $11 is generally the VTOC (a.k.a. disk catalog), and this disk has one sector that even the original RWTS can't read. Either I'm incredibly unlucky and got a bad original disk, or there's a secondary protection (nibble check) that reads that sector in a further non-standard way, or it's simply unused and is intentionally bad to make my life more difficult. The original disk has no problem booting or loading, so I'm guessing it's either a nibble check or an intentional glitch.

The disk's own RWTS gave no read errors on any other track. This is the power and the genius of Advanced Demuffin. Every disk must be able to read itself. So, let it read itself, then capture the data and write it out in a standard format.

Rebooting my work disk, I can now see the catalog on the demuffin'd copy.

```
]PR#5
...

]CATALOG,S6,D2

C1983 DSR^C#254
018 FREE

*A 010 LOGO
*A 009 BOOT
*A 019 CATS
*A 016 DONKEY
*A 014 DRAGON
*A 013 FLAMINGO
*A 010 IGUANA
*A 019 KANGAROOS
*A 016 KITTENS
*A 015 LLAMAS
*A 014 LOON
*A 017 MENU
*A 016 RHINOS
*A 012 SEAL
*A 015 SNAILS
*B 002 ARROWS2
*B 007 CAT.PICS
*B 009 DIRECTIONS.IMG
*B 009 DONKEY.PICS
*B 011 DOOR NUMBERS.IMG
*B 008 DRAGON.PICS
[...]
```

```
*B 002 ERROR STACK FIX
*B 006 FLAMINGO.PICS
*B 004 HOT WARM COLD
*B 008 IGUANA.PICS
*B 010 KANGAROO.PICS
*B 006 KITTEN.PICS
*B 006 LLAMA.PICS
*B 008 LOON.PICS
*B 002 NOT HERE
*B 007 PACK.OBJ
*B 002 PAT1
*B 002 PAT2
*B 002 PAT3
*B 002 PAT4
*B 002 PAT5
*B 002 PAT6
*B 002 PAT7
*B 002 PAT8
*B 010 RHINO.PICS
*B 010 SAFARI.DATA1
*B 009 SAFARI.DATA2
*B 012 SAFARI.DATA3
*B 032 SAFARI.OBJ
*B 009 SEAL.PICS
*B 007 SNAIL.PICS
*B 033 SUNBURST LOGO.PIC
*B 013 TITLE.PICS
*B 003 YES OR NO
*T 002 SET MEMLOC
 T 002 SOUND CHECK
```

This is encouraging, because it means that T11,S0F (the unreadable sector) is not used by the disk catalog.

No HELLO program, but that first file, LOGO, looks promising.

]RUN LOGO

OK, that does work. Good. That tells me that there aren't any callbacks in the program itself that are looking for a custom DOS. (I've seen several titles like this from other publishers. They PEEK or CALL specific locations that are intentionally different from standard DOS and reboot if things look amiss.)

I've cracked a number of Sunburst disks like this one. I've always gotten to this stage -- having converted the disk and confirmed that it works under standard DOS 3.3 -- then replaced tracks 0-2 with standard DOS 3.3 and declared victory. But I've never dug into the custom RWTS itself to see how it works.

For example, that table of non-standard address and data prologues, where every single track uses different parameters? That's unique to Sunburst. I've never seen anything like it anywhere else. Something somewhere is changing the RWTS on the fly with every disk read.

Anyway, partly because I'm curious and partly because I wanted to test my new Post-Demuffin Patcher utility (more on that in a minute), I decided to dig into the custom RWTS on this disk. Is there a small set of changes I could make to get my demuffin'd copy to run? Or do I really need to just wipe the entire DOS and replace it?

```
]PR#5
...

]BLOAD BOOT1,A$2600
]CALL -151

*FE89G FE93G      ; disconnect DOS

*B600<2600.2FFFM ; move RWTS into place
```

The RWTS is certainly shaped like a standard DOS 3.3 RWTS. Here's the code at $B94F, which is identical to the code I would find there on a freshly formatted DOS 3.3. (This is where it looks for the address prologue.)

```
*B94FL

B94F-   BD 8C C0    LDA   $C08C,X
B952-   10 FB       BPL   $B94F
B954-   C9 D5       CMP   #$D5
B956-   D0 F0       BNE   $B948
B958-   EA          NOP
B959-   BD 8C C0    LDA   $C08C,X
B95C-   10 FB       BPL   $B959
B95E-   C9 AA       CMP   #$AA
B960-   D0 F2       BNE   $B954
B962-   A0 03       LDY   #$03
B964-   BD 8C C0    LDA   $C08C,X
B967-   10 FB       BPL   $B964
B969-   C9 96       CMP   #$96
B96B-   D0 E7       BNE   $B954
```

But as I saw in the nibble editor, each track uses a different value for the third byte in the address prologue. Track 0 is the normal "D5 AA 96", but track 1 uses "D5 AA 97", track 2 uses "D5 AA 9A", &c. The code listed above will only read track 0. Something must be changing the value at $B96A before this code is run.

Turning to my trusty Disk Fixer sector editor, I search for the hex sequence "6A B9" and find one match on T00,S04. That sector is loaded at $BA00.
Turning back to the extracted boot1 code on my work disk, there appears to be a routine that starts at $BA69.

```
*BA69L

; save accumulator
BA69-   48          PHA

; phase number (track number x2)
BA6A-   A5 2A       LDA   $2A

; divided by 2, so now just a track
; number
BA6C-   4A          LSR
BA6D-   A8          TAY
BA6E-   B9 29 BA    LDA   $BA29,Y

; Aha! This is modifying the third byte
; of the address prologue, just as I
; suspected. And it's using the track
; number as an index into a table of
; some sort.
BA71-   8D 6A B9    STA   $B96A
```

```
; also changes the third byte of the
; address prologue on write
BA74-   8D 84 BC    STA   $BC84

; another table lookup
BA77-   B9 34 BA    LDA   $BA34,Y

; changing the third byte of the data
; prologue (read)
BA7A-   8D FC B8    STA   $B8FC

; ...and write
BA7D-   8D 5D B8    STA   $B85D
```

But wait! There's more!

```
; if track is $11...
BA80-   C0 11       CPY   #$11
BA82-   D0 03       BNE   $BA87

; then sector $0E is really sector $02
BA84-   A9 02       LDA   #$02

; (dummy opcode to hide next 2 bytes)
BA86-   AC

; otherwise sector $0E is itself
BA87-   A9 0E       LDA   #$0E
BA89-   8D C0 BF    STA   $BFC0
```

WTF.

In case you have no idea what's going on here (I've never seen anything like it)... The table from $BFB8..$BFC7 maps physical to logical sectors. (There's an identical table on T00,S00, but it's only used by boot0 for loading boot1. This one is used by the RWTS for the lifetime of the disk.) Anyway, this code is changing that table based on the track number. So, on track $11 (and only track $11), physical sector $08 is treated as sector $02. On all other tracks, physical sector $08 is treated as sector $0E.

No, I don't know why, except f--- you, that's why.

At any rate, this confirms my theory that there's some track-based lookup table being used to modify the expected address and data prologues on each track. Fun(*) fact: the table at $BA29 is part of the denibblization process that turns raw nibbles on disk into values in memory, and vice-versa. So they didn't need to "waste" 35 bytes to store custom address prologues for each track; they re-used a table that the RWTS uses for an entirely different purpose -- and one that comprises valid raw nibbles.

(*) not guaranteed, actual fun may vary

Turning back to Disk Fixer, I searched for "20 69 BA" (JSR $BA69) and found nothing. But searching for "4C 69 BA" (JMP $BA69) got two matches: one on T00,S00 and one on T00,S08. The one on T00,S08 looks more promising. That sector is loaded at $BE00.

"Beneath Apple DOS" (p. 8-39) says that the routine at $BE5A is "MYSEEK", and that it is called immediately before the RWTS moves the drive head to the appropriate track in preparation for a read or write. Furthermore, my sector search found the JMP instruction at $BE8B, which (according to "Beneath Apple DOS") is the final instruction of the "MYSEEK" routine.

Looking at a freshly formatted DOS 3.3 disk, it appears that the proper JMP is to $B9A0, which (not coincidentally) is the final instruction of the custom routine at $BA69, after it restores all registers and flags:

```
BA8C-   68          PLA
BA8D-   69 00       ADC   #$00
BA8F-   48          PHA
BA90-   AD 78 04    LDA   $0478
BA93-   90 2B       BCC   $BAC0
...
BAC0-   C9 22       CMP   #$22
BAC2-   69 00       ADC   #$00
BAC4-   8D 78 04    STA   $0478
BAC7-   68          PLA
BAC8-   4C A0 B9    JMP   $B9A0   <-- !
```

This patch should restore order to the universe:

```
T00,S08,$8C change 69BA to A0B9
```

But wait, there's more! I also spotted this difference, in the beginning of the routine to write the address field:

```
*BC69L

BC69-   4C B8 B6    JMP   $B6B8   <-- !
BC6C-   EA          NOP
BC6D-   EA          NOP
BC6E-   EA          NOP
BC6F-   9D 8D C0    STA   $C08D,X
BC72-   DD 8C C0    CMP   $C08C,X
BC75-   EA          NOP
BC76-   88          DEY
BC77-   D0 F0       BNE   $BC69
BC79-   A9 D5       LDA   #$D5
BC7B-   20 D5 BC    JSR   $BCD5
BC7E-   A9 AA       LDA   #$AA
BC80-   20 D5 BC    JSR   $BCD5
BC83-   A9 96       LDA   #$96
BC85-   20 D5 BC    JSR   $BCD5
```

On a normal DOS 3.3 disk, the code at $BC69 and $BC6C looks like this:

```
BC69-   20 C3 BC    JSR   $BCC3
BC6C-   20 C3 BC    JSR   $BCC3
```

$BCC3 is just an "RTS" instruction. The entire thing is a precise wait loop to write out self-sync bytes ($FF) before the address field. JSR always takes 6 cycles, and RTS also always takes 6 cycles. So the two instructions wait a total of 24 cycles before the "STA $C08D,X" instruction writes the sync byte to disk.

So what's at $B6B8? I'm glad you asked.

```
*B6B8L
                                        cycles
B6B8-   C0 05       CPY   #$05   ; 2
B6BA-   A9 00       LDA   #$00   ; 2
B6BC-   B0 06       BCS   $B6C4  ; 2(*)
B6BE-   EA          NOP          ; 2
B6BF-   A9 FF       LDA   #$FF   ; 2
B6C1-   C5 00       CMP   $00    ; 3
B6C3-   38          SEC          ; 2
B6C4-   B0 00       BCS   $B6C6  ; 3
B6C6-   4C 6F BC    JMP   $BC6F  ; 3
```

(*) The Y register is 6 the first time through this loop, so the branch at $B6BC will be taken the first two times, then not taken afterwards. When the branch is taken, it requires 3 CPU cycles and continues at $B6C4, so the total time is 2+2+3+3+3=13 cycles. When that branch is not taken, it requires 2 CPU cycles, falls through to $B6BE, and the whole thing takes 2+2+2+2+2+3+2+3+3=21 cycles. Adding 3 for the initial JMP at $BC69, the whole thing (starting at $BC69, up to but not including the STA at $BC6F) burns either 16 or 24 cycles.

Low-level disk activity is highly dependent on cycle-accurate counting (since the physical floppy disk keeps spinning whether you write to it or not). In a normal DOS, this wait routine (two consecutive JSR and RTS instructions) always burns exactly 24 cycles. On these disks, it sometimes burns 16 cycles, sometimes 24. So this RWTS writes the first 2 sync bytes faster than usual, then 4 sync bytes at the usual speed.

Also, the first two times, it writes $00 instead of $FF. I have no idea what effect that has on a physical floppy disk. All valid nibbles have the high bit set, so maybe it's just dead space? The entire thing is weird.

This patch should restore the original wait loop code at $BC69:

```
T00,S06,$69 change 4CB8B6EAEAEA
                 to 20C3BC20C3BC
```

All the other differences in this RWTS center around the address and data prologue and epilogue bytes. The RWTS apparently only uses a single epilogue byte ($97), so some of the branch instructions are subtly different. For example, checking the address epilogue:

```
*B98BL

B98B-   BD 8C C0    LDA   $C08C,X
B98E-   10 FB       BPL   $B98B
```

```
; usually $DE
B990-   C9 97        CMP   #$97
B992-   D0 AE        BNE   $B942
B994-   EA           NOP
B995-   BD 8C C0     LDA   $C08C,X
B998-   10 FB        BPL   $B995

; usually $AA
B99A-   C9 00        CMP   #$00

; usually BNE
B99C-   F0 A4        BEQ   $B942
B99E-   18           CLC
B99F-   60           RTS
```

All told, the RWTS needed 15 patches (including the two I mentioned already, and the three listed above) to be able to read and write a standard disk.

A lot of disks need this sort of post-demuffin patching, and I got tired of doing it manually, so I wrote a program to do it for me. It is called, unsurprisingly, Post-Demuffin Patcher. It prompts you to select a slot and drive, then reads the demuffin'd disk, checks for a modified DOS 3.3-shaped RWTS, and applies the necessary patches so the disk can read itself. (It can also detect and bypass some nibble checks.) I've included a copy of Post-Demuffin Patcher on my work disk; the full source code is currently available at <http://archive.org/details/PostDemuffinPatcher4am>.

Having already seen several Sunburst disks with identical protection, I knew it would be worth my time to add the detection and repair functions for this crazy RWTS to Post-Demuffin Patcher. So that's exactly what I did.

Now, whenever I come across a Sunburst disk (including this one), I can let Post-Demuffin Patcher do the work for me:

```
[S6,D1=demuffin'd copy]

]PR#5
...

]BRUN PDP

T00,S03,$40 change D0 to F0
T00,S03,$9C change F0 to D0
T00,S06,$69 change 4CB8B6EAEAEA to 20C3
C20C3BC
T00,S08,$8C change 69BA to A0B9
T00,S03,$91 change 97 to DE
T00,S03,$9B change 00 to AA
T00,S03,$35 change D3 to DE
T00,S03,$3F change 00 to AA
T00,S06,$AE change 97 to DE
T00,S06,$B3 change 00 to AA
T00,S06,$B8 change 00 to EB
T00,S02,$9E change D3 to DE
T00,S02,$A3 change 00 to AA
T00,S02,$A8 change 00 to EB
T00,S02,$AD change 00 to FF
```

(This is the actual output of the program. Post-Demuffin Patcher prints out the changes it is going to make before it writes them to the disk.)

The first 2 lines are fixing the non-standard branch instructions around the second epilogue byte. The next 2 lines are the 2 patches I found earlier. The rest are single-byte patches to get the RWTS to read and write a standard disk with "D5 AA 96"/"D5 AA AD" prologues and "DE AA EB" epilogues.

I should point out that Post-Demuffin Patcher is really quite conservative in making patches. It checks a lot of the surrounding code before deciding to patch a specific location (like the RWTS patches). In the case of changing the JMP at $BE8C from $BA69 to $B9A0, it actually checks every single byte of code at $BA69 to ensure that it recognizes the custom routine as the Sunburst RWTS. And there were dozens of patches that it didn't make to this disk, because it decided they weren't needed or it wasn't 100% sure what was going on.

```
]PR#6
...

And it crashes on boot.

9D86-    A=FD X=38 Y=1B P=33 S=FB
*
```

Wait, what?

After examining this disk -- which uses identical copy protection to dozens of other Sunburst disks -- I finally figured out what the difference was: the original disk uses disk volume 001 instead of the default (254). Why is this a problem? I'm glad you asked. I used Advanced Demuffin to convert the disk to a standard format, and part of that "standard format" included writing the data to a freshly formatted disk... which uses the default disk volume 254.

I could repeat the entire process and format my target disk as disk volume 001, but that will present another problem down the line. Ultimately, I want to distribute this as a .dsk image file, and .dsk image files don't have any way of storing the volume number. They are purely a sector dump; that's just the nature of the file format. The volume number is stored in the address field, but .dsk files don't store the address field. Emulators assume that every .dsk image uses disk volume 254.

OK, so if .dsk files don't store the volume number, then why does it matter? I'm glad you asked. Besides appearing in every sector's address field on a physical floppy disk, the volume number is stored in three different places when a disk is initialized:

1. $B7EB (T00,S01,$EB), in the RWTS parameter table used by boot1 to load DOS from tracks 0-2 [“Beneath Apple DOS”, p. 8-35]

2. $AA66 (T01,S09,$66), in the parsed keyword table used by DOS to load the startup program (and every other file loaded after that) [ibid., p. 8-21]

3. $B3C1 (T11,S00,$06), in the VTOC header [ibid., p. 8-32]

My (non-working) copy has a $01 in each of those locations. Because of this mismatch (specifically the first 1, on T00,S01), boot1 fails to load boot2. Literally every sector read fails. Then it jumps to boot2 at $9D84, which was never loaded, so it crashes. Which is exactly the behavior I saw.

Using my trusty Disk Fixer sector editor, I changed each of those locations to $FE.

T00,S01,$EB change 01 to FE
T01,S09,$66 change 01 to FE
T11,S00,$06 change 01 to FE

Success! The copy finally boots and runs. There doesn't appear to be any further copy protection. And because the DOS on my copy now expects disk volume 254, it also works when I create a .dsk image file out of it and boot it in an emulator.

(Note to self: add this to a future version of Post-Demuffin Patcher.)

Quod erat liberandum.