IMPOSSIBLE MISSION-II

2014-09-08

4am
to deprotect
and preserve

Contents

---------Impossible Mission II---------
A 4am crack   2014-09-08

"Impossible Mission II" is a 1988 action/adventure game developed by FACS Entertainment Software and distributed by Epyx. Programming by Douglas D. Dragin, Howard E. Scheer, Richard T. Unruh, and Tom Zerucha. Graphics by Douglas D. Dragin, Michael L. Snyder, and Tom Zerucha.

[The copy protection is identical to "The Movie Monster Game," also distributed by Epyx. This write-up starts out quite similar to that one, then it goes off into the weeds a bit and comes out the other side refreshed and beaming and I have no idea what I'm saying right now so let's just get on with it.]

The game comes on a double-sided floppy disk and uses both sides. Side B appears unprotected; COPYA can copy it without an issue. Side A, on the other hand, fails hard and fast in COPYA. EDD 4 bit copy gives no read errors, but the copy it produces just reboots endlessly.

Firing up my trusty Copy][+ sector editor, I press "P" to get to the Sector Editor Patcher and selected "DOS 3.3 Patched." This option ignores the epilogue bytes after the address field and data field (normally "DE AA EB"). It doesn't work on every disk; lots of disks use non-standard prologue bytes as well, or don't use 16 sectors, or any one of a hundred different ways to deviate from the norm within the tolerance of the Disk II hardware. But it works with this one! Ignoring epilogue bytes, the sector editor can whiz through the entire disk and read all the data from every sector on every track.

Based on my limited experience cracking other disks, I would guess that this disk has

- Standard prologue bytes before the address and data fields [otherwise Copy][+ sector editor would give read errors, even with the "DOS 3.3 PATCHED" option]

- Non-standard epilogue bytes after the address and data fields [otherwise COPYA would work]

- Some secondary protection [otherwise the bit copy created with EDD 4 would work]

Given the (relatively) weak structural protection, I used to turn to the DOS 3.3 master disk, patch the RWTS to ignore checksums and epilogue bytes (changing $B942 from "SEC" to "CLC"), and run COPYA. Then, one fine day, and completely by accident, I came across an original disk with a bad sector. I suppose this shouldn't surprise me. These floppies are decades old by now; it's amazing any of them work at all.

The point is, I shouldn't be using tools that ignore potentially serious read errors. So, no more COPYA+B942:18 patch. From now on, it's Super Demuffin or Advanced Demuffin to convert disks to a standard format.

The original disk sounds a lot like a DOS 3.3 disk (reading track 2, then 1, then 0, then swinging out to a higher track). However, I don't see any disk catalog on track $11 or any other clues about how this disk is organized.

Time for boot tracing with AUTOTRACE.

```
[S6,D1=original disk, side A]
[S5,D1=my work disk]

]PR#5
CAPTURING BOOT0
...reboots slot 6...
...reboots slot 5...
SAVING BOOT0
```

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.

```
]CALL -151

*800<2800.28FFM

*801L
...
```

Everything here looks pretty normal (i.e. just like an unprotected DOS 3.3 disk), until it goes to jump to the boot1 code. Usually that happens with an indirect JMP ($08FD), which, in a normal boot0, will end up continuing execution at $B700 which is stored in track 0, sector 1. But in this case, I see:

```
084A-   4C 00 BB    JMP   $BB00
```

Highly suspect. I definitely want to see what evil lurks at $BB00. That area of memory is normally reserved for the denibblizing process when reading data from a sector. It's scratch space, essentially. It's overwritten every time the disk reads itself (after boot1 is loaded).

But $BB00 isn't loaded yet, because I interrupted the boot process before it could be loaded. So now I need to trace the boot again, but a little bit further -- far enough for boot0 to load boot1 (including the suspicious code at $BB00), but no further.

My work disk has another program, unimaginatively named AUTOTRACE1, which does just that. It loads track 0, sector 0, then patches the boot0 code at $084A to call back to a routine under my control (instead of jumping to the original disk's boot1 code).

```
]BRUN AUTOTRACE1
CAPTURING BOOT1
...reboots slot 6...
...reboots slot 5...
SAVING BOOT1
```

Let's see what we have.

```
]BLOAD BOOT1,A$2600
]CALL -151

*FE89G FE93G     ; disconnect DOS

*B600<2600.2FFFM ; move RWTS into place

*BB00L

; initialize some zero page addresses
BB00-   A9 00       LDA   #$00
BB02-   A2 F0       LDX   #$F0
BB04-   9A          TXS
BB05-   95 00       STA   $00,X
BB07-   E8          INX
BB08-   D0 FB       BNE   $BB05
BB0A-   A9 0A       LDA   #$0A
BB0C-   85 FC       STA   $FC
```

```
; Turn on the disk motor. Zero page $2B
; contains the slot number x 16. This
; is the standard way to access low-
; level disk commands.
BB0E-   A6 2B       LDX   $2B
BB10-   BD 89 C0    LDA   $C089,X
BB13-   BD 8E C0    LDA   $C08E,X

; a counter of some kind
BB16-   A9 80       LDA   #$80
BB18-   85 FD       STA   $FD
BB1A-   C6 FD       DEC   $FD

; $BB98 is what I call "The Badlands"
; i.e. the code from which there is
; no return. Skipping ahead, I can see
; that $BB98 is one branch away from
; tweaking the reset vector and
; rebooting the computer. Which is,
; you know, not what we want.
BB1C-   F0 7A       BEQ   $BB98

; $BBA5 looks for the standard address
; prologue, as a setup for the real
; test which comes next.
BB1E-   20 A5 BB    JSR   $BBA5

; If, for some reason, that doesn't
; work, off to The Badlands with you.
BB21-   B0 75       BCS   $BB98
```

```
; Search for a specific sequence of
; nibbles in the "dead zone" between
; the address field and data field.
; This area is normally not important,
; so COPYA didn't copy it precisely
; because normal disks don't care.
; (Actually, it's even more evil than
; that, because the original disk is
; written with timing bits in specific
; non-standard places between the
; nibbles in the dead zone. This code
; not only requires the right nibbles
; in the right order, it reads them
; just slightly slower than normal. So
; the timing bits need to be in the
; right places too, or the disk will
; get out of sync and read the wrong
; nibble values. This will trip up even
; the best bit copiers. And you can
; forget about making a disk image for
; emulators -- those don't store timing
; bits at all.)
BB23-   A5 F9       LDA   $F9
BB25-   C9 08       CMP   #$08
BB27-   D0 F1       BNE   $BB1A
BB29-   A0 00       LDY   #$00
BB2B-   BD 8C C0    LDA   $C08C,X
BB2E-   10 FB       BPL   $BB2B
BB30-   88          DEY

; off to The Badlands
BB31-   F0 65       BEQ   $BB98
BB33-   C9 D5       CMP   #$D5
BB35-   D0 F4       BNE   $BB2B
BB37-   A0 00       LDY   #$00
BB39-   BD 8C C0    LDA   $C08C,X
BB3C-   10 FB       BPL   $BB39
BB3E-   88          DEY
```

```
; off to The Badlands
BB3F-   F0 57        BEQ   $BB98
BB41-   C9 E7        CMP   #$E7
BB43-   D0 F4        BNE   $BB39
BB45-   BD 8C C0     LDA   $C08C,X
BB48-   10 FB        BPL   $BB45
BB4A-   C9 E7        CMP   #$E7

; off to The Badlands
BB4C-   D0 4A        BNE   $BB98
BB4E-   BD 8C C0     LDA   $C08C,X
BB51-   10 FB        BPL   $BB4E
BB53-   C9 E7        CMP   #$E7

; off to The Badlands
BB55-   D0 41        BNE   $BB98

; kill some time to get out of sync
; with the "proper" start of nibbles)
BB57-   BD 8D C0     LDA   $C08D,X
BB5A-   A0 10        LDY   #$10
BB5C-   24 80        BIT   $80

; now start looking for nibbles that
; don't really exist (except they do,
; because we're out of sync and reading
; timing bits as data)
BB5E-   BD 8C C0     LDA   $C08C,X
BB61-   10 FB        BPL   $BB5E
BB63-   88           DEY

; off to The Badlands
BB64-   F0 32        BEQ   $BB98
BB66-   C9 EE        CMP   #$EE
BB68-   D0 F4        BNE   $BB5E
BB6A-   EA           NOP
BB6B-   EA           NOP
```

```
; store nibble sequence that follows
BB6C-   A0 07       LDY   #$07
BB6E-   BD 8C C0    LDA   $C08C,X
BB71-   10 FB       BPL   $BB6E
BB73-   99 F0 00    STA   $00F0,Y
BB76-   EA          NOP
BB77-   88          DEY
BB78-   10 F4       BPL   $BB6E

; wait, it gets better -- this disk
; actually uses the raw nibble data it
; stores in this "dead zone" AS THE
; DECRYPTION KEY FOR THE REST OF BOOT1
BB7A-   A2 03       LDX   #$03
BB7C-   A9 00       LDA   #$00
BB7E-   A8          TAY
BB7F-   85 F8       STA   $F8
BB81-   A9 B7       LDA   #$B7
BB83-   85 F9       STA   $F9
BB85-   B5 F0       LDA   $F0,X
BB87-   51 F8       EOR   ($F8),Y
BB89-   91 F8       STA   ($F8),Y
BB8B-   88          DEY
BB8C-   D0 F7       BNE   $BB85
BB8E-   E6 F9       INC   $F9
BB90-   CA          DEX
BB91-   10 F2       BPL   $BB85

; now that the boot1 code is decrypted,
; jump to it as normal
BB93-   A6 2B       LDX   $2B
BB95-   4C 00 B7    JMP   $B700
```

```
; The Badlands
BB98-   C6 FC        DEC   $FC
BB9A-   F0 03        BEQ   $BB9F
BB9C-   4C 16 BB     JMP   $BB16
BB9F-   EE F4 03     INC   $03F4
BBA2-   6C FC FF     JMP   ($FFFC)

; subroutine (called from $BB1E) to
; find the next address prologue and
; skip over the address field to
; position the drive head to read the
; special nibbles in the dead zone
BBA5-   A0 FD        LDY   #$FD
BBA7-   84 F0        STY   $F0
BBA9-   C8           INY
BBAA-   D0 04        BNE   $BBB0
BBAC-   E6 F0        INC   $F0
BBAE-   F0 3D        BEQ   $BBED
BBB0-   BD 8C C0     LDA   $C08C,X
BBB3-   10 FB        BPL   $BBB0
BBB5-   C9 D5        CMP   #$D5
BBB7-   D0 F0        BNE   $BBA9
BBB9-   EA           NOP
BBBA-   BD 8C C0     LDA   $C08C,X
BBBD-   10 FB        BPL   $BBBA
BBBF-   C9 AA        CMP   #$AA
BBC1-   D0 F2        BNE   $BBB5
BBC3-   A0 03        LDY   #$03
BBC5-   BD 8C C0     LDA   $C08C,X
BBC8-   10 FB        BPL   $BBC5
BBCA-   C9 96        CMP   #$96
BBCC-   D0 E7        BNE   $BBB5
[...]
```

```
BBCE-   A9 00       LDA   #$00
BBD0-   85 F1       STA   $F1
BBD2-   BD 8C C0    LDA   $C08C,X
BBD5-   10 FB       BPL   $BBD2
BBD7-   2A          ROL
BBD8-   85 F0       STA   $F0
BBDA-   BD 8C C0    LDA   $C08C,X
BBDD-   10 FB       BPL   $BBDA
BBDF-   25 F0       AND   $F0
BBE1-   99 F8 00    STA   $00F8,Y
BBE4-   45 F1       EOR   $F1
BBE6-   88          DEY
BBE7-   10 E7       BPL   $BBD0
BBE9-   A8          TAY
BBEA-   EA          NOP
BBEB-   18          CLC
BBEC-   60          RTS
BBED-   38          SEC
BBEE-   60          RTS
```

Because the next stage of the boot is encrypted, I can't simply bypass this copy protection routine. I need to let it run at least once, from the original disk, so that it can decrypt the rest of the code. The decryption happens at $BB7A..$BB92, then at $BB95 it jumps to the start of boot1 ($B700, just like a normal DOS 3.3 disk). That's where I need to interrupt the boot so I can capture the decrypted boot1 code before the game continues.

Except...

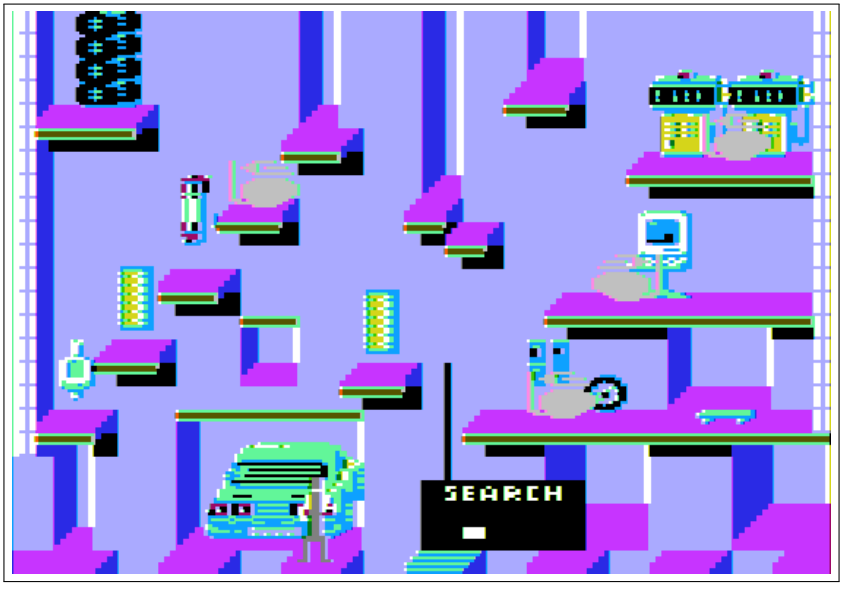
This is all very interesting, but I've seen it before. Epyx uses this exact nibble check on several of their games. And they're not the only ones. I've seen the same pattern (boot0 jumps to a nibble check at $BB00, which then jumps to boot1) on dozens of disks.

Computers are good at automating repetitive tasks, right? Let's automate this one.

~

My AUTOTRACE program is a combination of Applesoft BASIC (HELLO) and assembly language (AUTOTRACE0 and AUTOTRACE1, each of which traced a different phase of the boot process). Now that I'm getting fancy, I've decided to combine the two binary programs into one (now called AUTOTRACE with no suffix).

The HELLO program checks for the presence of a marker in low memory. (More on this later.) If found, it interprets the marker as a command and branches accordingly. If this marker is not present, it checks for the presence of a BOOT0 file. If found, it just displays a CATALOG and stops. (What, you thought my boot tracing efforts always worked on the first try? Ha!) If there is no BOOT0 file, it BRUNs the AUTOTRACE program to try to capture the boot0 code from track 0, sector 0.



```
 30  ONERR  GOTO 90
 40  PRINT  CHR$ (4)"BLOAD BOOT0,
     A$9800"
 50  PRINT  CHR$ (4)"CATALOG"
 60  GOTO 1000
 90  REM  FIRST RUN
 91  POKE 216,0: REM  CLEAR ONERR

 94  PRINT "CAPTURING BOOT0"
 95  PRINT  CHR$ (4)"BRUN AUTOTRA
     CE"
```

The AUTOTRACE binary starts at $9600. $9600..$96F7 is copied from $C600, the disk controller ROM routine. (Since cards can be in any slot, this code has no hard-coded addresses and can be run from any page. Hooray for good design!) My code starts at $96F8.

```
; jump to boot0 capture (default)
96F8-   4C FB 96     JMP   $96FB

; relocate boot0 code to graphics page
; so it will survive a reboot
96FB-   A0 00        LDY   #$00
96FD-   B9 00 08     LDA   $0800,Y
9700-   99 00 28     STA   $2800,Y
9703-   C8           INY
9704-   D0 F7        BNE   $96FD

; set a marker in lower memory (the
; HELLO program will use this to
; determine what it should do next)
9706-   A9 80        LDA   #$80
9708-   8D 00 01     STA   $0100
970B-   49 A5        EOR   #$A5
970D-   8D 01 01     STA   $0101
```

```
; turn off the slot 6 drive motor
9710-   AD E8 C0    LDA   $C0E8

; reboot to my work disk
9713-   4C 00 C5    JMP   $C500
```

When that reboots, it will re-run the HELLO program on my work disk. This is where that marker in low memory comes into play. The HELLO program looks at the two bytes at $0100 (the "command" byte) and $0101 (the "checksum" byte) and branches accordingly.

```
 10 CMD =  PEEK (256)
 11 CHECKSUM =  PEEK (257)
 12  IF CMD = 128 AND CHECKSUM =
     37 THEN 100: REM  SAVE BOOT0
.
.
.
 100  REM  BOOT0 WAS CAPTURED, NO
     W SAVE IT
 101  REM
 105  PRINT "SAVING BOOT0"
 110  PRINT  CHR$ (4)"BSAVE BOOT0
     ,A$2800,L$100"
```

The next step is to try to capture the boot1 code, by patching boot0 at $084A so it jumps to my code instead of the boot1 code that it loaded from track 0. Previously, my checks for whether it was safe to patch $084A were very conservative. The instruction at $084A had to be exactly "6C FD 08", which is "JMP ($08FD)". This is why AUTOTRACE would stop after boot0 on disks like this one, because $084A jumps to $BB00 to perform the nibble check.

Instead of just stopping and making me look at the code and continue with the boot1 capture manually, let's get a little bit smarter. I'll add an explicit check for "JMP $BB00" and print a warning, then continue with the boot1 capture anyway. (It also handles "JMP ($BBFE)", which is a variation I've seen before on several disks.)

```
 114  IF  PEEK (10314) = 108 AND
      PEEK (10315) = 254 AND  PEEK
     (10316) = 187 THEN  PRINT "/
     !\ BOOT0 JUMPS TO ($BBFE)": GOTO
     130
 115  IF  PEEK (10314) = 76 AND  PEEK
     (10315) = 0 AND  PEEK (10316
     ) = 187 THEN  PRINT "/!\ BOO
     T0 JUMPS TO $BB00": GOTO 130

 120  IF  PEEK (10314) <  > 108 OR
      PEEK (10315) <  > 253 OR  PEEK
     (10316) <  > 8 THEN 1000
```

If the HELLO program decides that the boot0 code is standard enough that it can successfully patch it and capture boot1, it continues on line 130.

```
 130  REM  STANDARD BOOT0, SO
 131  REM  AUTOTRACE BOOT1
 132  REM
 135  PRINT "CAPTURING BOOT1"
 140  PRINT  CHR$ (4)"BLOAD AUTOT
      RACE"
 150  POKE 38649,22: POKE 38650,1
      51: CALL 38400: END
```

The POKEs on line 150 set up the jump at $96F8 to call my boot1 capture routine, which starts at $9716. The CALL executes $9600, which contains a copy of the disk controller ROM routine originally located at $C600. (As soon as the boot trace starts at $9600, it will overwrite the BASIC program in memory, so whatever setup I need to do from BASIC, I need to do it all at once before CALLing $9600. Having a configurable JMP at $96F8 allows me to store all the different trace routines in a single binary file.)

```
; this was set up by POKEs on line 150
96F8-   4C 16 97    JMP   $9716
.
.
.
; patch boot0 to jump to a routine
; under my control instead of
; continuing with boot1
9716-   A9 4C       LDA   #$4C
9718-   8D 4A 08    STA   $084A
971B-   A9 34       LDA   #$34
971D-   8D 4B 08    STA   $084B
9720-   A9 97       LDA   #$97
9722-   8D 4C 08    STA   $084C

; set up later memory move based on
; parameters from boot0 --
; $08FE contains the high byte of the
; starting address of boot1
9725-   AD FE 08    LDA   $08FE
9728-   8D 3A 97    STA   $973A

; $08FF contains the number of sectors
; in boot1
972B-   AD FF 08    LDA   $08FF
972E-   8D 35 97    STA   $9735

; start the boot
9731-   4C 01 08    JMP   $0801
```

```
; callback is here --
; relocate the entire boot1 code to the
; graphics page so it survives a reboot
9734-   A2 09       LDX   #$09
9736-   A0 00       LDY   #$00
9738-   B9 00 B6    LDA   $B600,Y
973B-   99 00 20    STA   $2000,Y
973E-   C8          INY
973F-   D0 F7       BNE   $9738
9741-   EE 3A 97    INC   $973A
9744-   EE 3D 97    INC   $973D
9747-   CA          DEX
9748-   10 EE       BPL   $9738

; set a marker in lower memory (the
; HELLO program will use this to
; determine what it should do next)
974A-   A9 81       LDA   #$81
974C-   8D 00 01    STA   $0100
974F-   49 A5       EOR   #$A5
9751-   8D 01 01    STA   $0101

; turn off the slot 6 drive motor
9754-   AD E8 C0    LDA   $C0E8

; reboot to my work disk
9757-   4C 00 C5    JMP   $C500
```

Once again, this will reboot to my work disk and run the HELLO program. This time, it will see a different marker and branch accordingly.

```
 13  IF CMD = 129 AND CHECKSUM =
     36 THEN 200: REM  SAVE BOOT1
.
.
.
 200  REM  BOOT1 WAS CAPTURED, NO
     W SAVE IT
 205  PRINT "SAVING BOOT1"
 210  PRINT  CHR$ (4)"BSAVE BOOT1
     ,A$2000,L$A00"
```

To determine if this boot1 code contains a "DOS 3.3-shaped" RWTS, I have a subroutine (starting at line 1200) that spot-checks a few critical locations. If those pass, I save the RWTS to its own file.

```
 220 RWTS = 0: GOSUB 1200
 230  IF RWTS = 0 THEN 400
 250  PRINT "SAVING RWTS"
 260  PRINT  CHR$ (4)"BSAVE RWTS,
     A$2200,L$800"
.
.
.
```

```
1200  REM   CHECK IF BOOT1 CONTA
    INS A NORMAL-SHAPED RWTS
1201  REM  (RWTS=1 ON EXIT IF FO
    UND)
1210 RWTS = 0
1219  REM   "STY $48: STA $49" A
    T $BD00?
1220  IF  PEEK (9984) <  > 132 THEN
     RETURN
1221  IF  PEEK (9985) <  > 72 THEN
     RETURN
1222  IF  PEEK (9986) <  > 133 THEN
     RETURN
1223  IF  PEEK (9987) <  > 73 THEN
     RETURN
1229  REM   "SEC: RTS" AT $B942?

1230  IF  PEEK (9026) <  > 56 THEN
     RETURN
1231  IF  PEEK (9027) <  > 96 THEN
     RETURN
1239  REM   "LDA $C08C,X" AT $B9
    4F?
1240  IF  PEEK (9039) <  > 189 THEN
     RETURN
1241  IF  PEEK (9040) <  > 140 THEN
     RETURN
1242  IF  PEEK (9041) <  > 192 THEN
     RETURN
1243  REM   "JSR $XX00" AT $BDB9

1244  IF  PEEK (10169) <  > 32 THEN
     RETURN
1245  IF  PEEK (10170) <  > 0 THEN
     RETURN
1250 RWTS = 1
1260  RETURN
```

Now the real fun begins. I've already printed a warning if boot0 jumps to $BB00. But what evil lurks at $BB00? I have the code in memory. I could scan it and find out.

Every nibble check I've ever seen at $BB00 has included the instruction "LDA $C089,X", which manually turns on the drive motor of the slot given in the X register (x16). It's just a convention. You could just as easily calculate the exact address and place it into the following instruction, like "LDA $C0E9" (without the X index). Or "LDX $C0E9". Or even "LDY $C089,X". The disk drive doesn't care. But I've never seen *any* variation here.)

So, let's see if we can find evidence of a nibble check, by scanning for that instruction. (Keep in mind that, on an unprotected DOS 3.3 disk, there isn't *any* executable code in the $BB00 page. So there's no chance of a false positive.)

```
 400  REM  NIBBLE CHECK AT $BB00?

 401  REM  (SCAN FOR LDA $C089,X)

 410 X = 9472
 420  IF  PEEK (X) <  > 189 THEN
     490
 430  IF  PEEK (X + 1) <  > 137 THEN
     490
 440  IF  PEEK (X + 2) <  > 192 THEN
     490
 450  PRINT "⁄!⁄ NIBBLE CHECK AT
     $BB00"
 460  GOTO 500
 490 X = X + 1: IF X < 9728 THEN
     420
 499  GOTO 1000
```

If it finds an "LDA $C089,X" instruction, it prints a warning and continues at line 500, where it scans for two more instructions. This disk (and others like it) have the following decryption loop after the nibble check:

```
BB85-   B5 F0       LDA   $F0,X
BB87-   51 F8       EOR   ($F8),Y
BB89-   91 F8       STA   ($F8),Y
BB8B-   88          DEY
BB8C-   D0 F7       BNE   $BB85
BB8E-   E6 F9       INC   $F9
BB90-   CA          DEX
BB91-   10 F2       BPL   $BB85
```

Looking through previous cracks of similar disks, it appears that they all use (F8),Y in their decryption loop. Again, there's no particular reason why I should look for that particular address, except that all these disks happen to use it.

```
 500  REM  ENCRYPTED BOOT1?
 501  REM   (SCAN FOR EOR ($F8),Y
     ; STA ($F8),Y IN $BB00 RANGE
     )
 510 X = 9472
 520  IF  PEEK (X) <  > 81 THEN 5
     90
 530  IF  PEEK (X + 1) <  > 248 THEN
     590
 540  IF  PEEK (X + 2) <  > 145 THEN
     590
 550  IF  PEEK (X + 3) <  > 248 THEN
     590
 560  PRINT "/!\ BOOT1 IS ENCRYPT
     ED"
 570  GOTO 600
 590 X = X + 1: IF X < 9728 THEN
     520
 599  GOTO 1000
```

If it finds the "EOR ($F8),Y" followed by "STA ($F8),Y", it prints a warning and continue to line 600, where I will try to patch the nibble check and trace the boot long enough for the original disk to decrypt itself.

I've seen two variations, even among Epyx disks, in how they continue to the real boot1 code after decryption. Some disks use "JMP ($08FD)" like DOS 3.3 does; other disks use "JMP $B700". I'll need to check for both variants.

```
 600  REM  TRY TO AUTO-DECRYPT BO
     OT1
 601  REM  (SCAN FOR JMP ($08FD)
     OR JMP $B700)
 602  REM  (IF FOUND, PATCH IT AN
     D REBOOT)
 610 X = 9472
 620  IF  PEEK (X) <  > 76 AND  PEEK
     (X) <  > 108 THEN 690
 630  IF  PEEK (X + 1) <  > 0 AND
      PEEK (X + 1) <  > 253 THEN
     690
 640  IF  PEEK (X + 2) <  > 183 AND
      PEEK (X + 2) <  > 8 THEN 69
     0
 650  PRINT "DECRYPTING BOOT1"
 651  PRINT  CHR$ (4)"BLOAD AUTOT
     RACE"
 655 X = X - 9472
 656  POKE 38767,x
 657  POKE 38772,X + 1
 658  POKE 38777,X + 2
 660  POKE 38649,90: POKE 38650,1
     51: CALL 38400: END
 690 X = X + 1: IF X < 9728 THEN
     620
 699  GOTO 1000
```

If it finds either "JMP ($08FD)" or "JMP $B700" in the $BB00 range, I BLOAD the AUTOTRACE binary (line 651) and patch it with the location of the JMP instruction, so that it can patch the post-nibble-check jump to call back to a routine under my control. Then I patch the JMP at $96F8 and call $9600.

This is a lot of indirection, so here's the specifics, using this disk as an example. I know from my previous manual inspection that this disk has a nibble check at $BB00 and a decryption loop at $BB85, then it jumps directly to $B700 at $BB95. Therefore, I need to create (and run) a boot tracing program that

1. loads boot0
2. patches $084A to jump to a routine under my control (instead of jumping to the nibble check at $BB00)
3. jumps to boot0
4. in its first callback routine, patches $BB95 to call a second routine under my control (instead of jumping to the decrypted boot1 code at $B700)
5. jumps to $BB00
6. in its second callback routine, captures the decrypted boot1 code
7. saves the decrypted boot1 code to disk

Line 651 BLOADs the AUTOTRACE binary, but it still needs some customization. Lines 655-659 set up the patch based on the address of the jump to boot1 after the nibble check ($BB95). Line 660 POKEs the jump address at $96F8 and calls $9600 (step 1).

```
96F8-   4C 5A 97    JMP   $975A
.
.
.
; patch boot0 (step 2)
975A-   A9 4C       LDA   #$4C
975C-   8D 4A 08    STA   $084A
975F-   A9 6C       LDA   #$6C
9761-   8D 4B 08    STA   $084B
9764-   A9 97       LDA   #$97
9766-   8D 4C 08    STA   $084C

; call boot0 (step 3)
9769-   4C 01 08    JMP   $0801

; first callback -- patch post-nibble-
; check jump to boot1 (step 4)
976C-   A9 4C       LDA   #$4C

; this address varies from disk to disk
; so my Applesoft program POKEs it
; (line 656)
976E-   8D 95 BB    STA   $BB95
9771-   A9 7E       LDA   #$7E

; address varies, POKEd at line 657
9773-   8D 96 BB    STA   $BB96
9776-   A9 97       LDA   #$97
```

```
; address varies, POKEd at line 658
9778-   8D 97 BB    STA   $BB97

; call nibble check (step 5)
977B-   4C 00 BB    JMP   $BB00

; second callback -- capture decrypted
; boot1 code by moving it to
977E-   A2 0A       LDX   #$0A
9780-   A0 00       LDY   #$00
9782-   B9 00 B6    LDA   $B600,Y
9785-   99 00 20    STA   $2000,Y
9788-   C8          INY
9789-   D0 F7       BNE   $9782
978B-   EE 84 97    INC   $9784
978E-   EE 87 97    INC   $9787
9791-   CA          DEX
9792-   D0 EE       BNE   $9782

; set a marker in lower memory (the
; HELLO program will use this to
; determine what it should do next)
9794-   A9 82       LDA   #$82
9796-   8D 00 01    STA   $0100
9799-   49 A5       EOR   #$A5
979B-   8D 01 01    STA   $0101

; turn off the slot 6 drive motor
979E-   AD E8 C0    LDA   $C0E8

; reboot to my work disk
97A1-   4C 00 C5    JMP   $C500
```

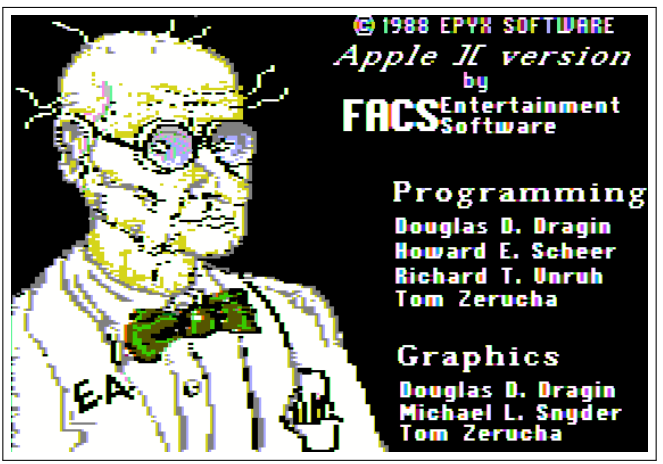
Once this runs, it will reboot my work disk, which will notice the command in $0100 and branch to line 700 to save the decrypted boot1 code.

```
 700  REM  DECRYPTED BOOT1 WAS CA
     PTURED, NOW SAVE IT
 710  PRINT "SAVING BOOT1 DECRYPT
     ED"
 720  PRINT  CHR$ (4)"BSAVE BOOT1
      DECRYPTED,A$2000,L$A00"
```

Now that the boot1 code is decrypted, I need to re-run the RWTS check to see if I can find a "DOS 3.3-shaped" RWTS after all that nonsense.

```
 730 RWTS = 0: GOSUB 1200
 740  IF RWTS THEN 250
 750  GOTO 1000
```

~



Starting with a fresh work disk with this updated AUTOTRACE program, here's what it looks like when I unleash it on "Impossible Mission II":

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

]PR#5
CAPTURING BOOT0
...reboots slot 6...
...reboots slot 5...
SAVING BOOT0
/!\ BOOT0 JUMPS TO $BB00
CAPTURING BOOT1
...reboots slot 6...
...reboots slot 5...
SAVING BOOT1
/!\ NIBBLE CHECK AT $BB00
/!\ BOOT1 IS ENCRYPTED
DECRYPTING BOOT1
...reboots slot 6...
...reboots slot 5...
SAVING BOOT1 DECRYPTED
SAVING RWTS
```

```
]CATALOG

C1983 DSR^C#254
272 FREE

 A 013 HELLO
 B 003 AUTOTRACE
 B 024 ADVANCED DEMUFFIN 1.5
 T 147 ADVANCED DEMUFFIN 1.5 DOCS
 B 003 BOOT0
 B 012 BOOT1
 B 012 BOOT1 DECRYPTED
 B 010 RWTS

]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]
```

Kick. Ass.

Now I have a copy of each disk in a standard format that can be read by any third-party tool. (Side B was already in a standard format.) But side A won't boot yet, because it still has the original (encrypted) boot1 code on track 0, with the original (encrypted) RWTS as well.

To write the decrypted boot1 to disk, I wrote a short program.

```
08C0-   A9 08       LDA   #$08
08C2-   A0 E8       LDY   #$E8
08C4-   20 D9 03    JSR   $03D9
08C7-   AC ED 08    LDY   $08ED
08CA-   88          DEY
08CB-   10 05       BPL   $08D2
08CD-   A0 0F       LDY   #$0F
08CF-   CE EC 08    DEC   $08EC
08D2-   8C ED 08    STY   $08ED
08D5-   CE F1 08    DEC   $08F1
08D8-   CE E1 08    DEC   $08E1
08DB-   D0 E3       BNE   $08C0

08E0- 17 0A 0A 1B E8 B7 00 B4
         ^^
         ++-- write 10 sectors (0-9)

08E8- 01 60 01 00 00 09 FB 08
                  ^^ ^^
       start on --++ ++-- start on
       track 0           sector 9

08F0- 00 2F 00 00 02 00 FE 60
      ^^^^^
      +++++-- start at address $2F00

08F8- 01 00 00 00 01 EF D8 00

*BSAVE WRITE BOOT1 DECRYPTED,A$8C0,L$40

*BLOAD BOOT1 DECRYPTED,A$2600

*8C0G
```

Turning to my trusty Copy][+ sector editor, I can remove the JMP-to-nibble-check-and-decryption-loop after boot0. The disk now has a decrypted version of boot1, and I don't ever want to run the nibble check.

T00,S00,$4C change "BB" to "B7"

Success! The game boots and runs with no complaint. The RWTS appears to be liberal enough that it can read my copy without modification, and there doesn't appear to be any further protection.

Quod erat liberandum.

EOF