Ancient Legends

2016-09-08

4am to deprotect and preserve

Contents

```
------------Ancient Legends------------
A 4am crack                  2016-09-08
-------------------. updated 2015-09-09
                   |___________________

Name: Ancient Legends
Version: 1.0
Genre: adventure
Year: 2016
Authors: Martin Haye, Brendan Robert,
  Dave Schmenk, Andrew Hogan, Seth
  Sternberger
Publisher: none(*)
Platform: Apple //e or later (128K)
Media: double-sided 5.25-inch floppy
OS: ProDOS 1.9
Previous cracks: none
```

```
(*) The game was released at Kansasfest
    2016. Only five physical copies
    were ever produced.
```

Chapter 0
In Which Various Automated Tools Fail In Interesting Ways

```
COPYA
  read error on final pass
  copy boots to title screen then exits
  to BASIC prompt with no OS loaded

Locksmith Fast Disk Backup
  read error on T22,S01

EDD 4 bit copy (no sync, no count)
  works
```

```
Copy ][+ nibble editor
  the unreadable T22,S01 does exist,
  but the data field looks corrupted
  data prologue = "D5 AF FC" ?

                 --v--

TRACK: 22  START: 1800  LENGTH: 3DFF

2B80: 96 96 96 96 96 96 96 96   VIEW
2B88: 96 96 96 DE AA EB DA B4
2B90: B4 B5 FE FF FF FF FF FF
2B98: FF FF FF FF FF FF FF FF
2BA0: D5 AA 96 FF FE BB AA AE  <-2BA7
      ^^^^^^^^       ^^^^^ ^^
  address prologue   T=$22

2BA8: AF EA FB DE AA EB FF FF
      ^^       ^^^^^^^^
   S=$01   address epilogue

2BB0: FF FF FF FF FF FF FF FF
2BB8: D5 AF FC FE E7 E7 E7 E7
      ^^^^^^^^
   data prologue?

2BC0: E7 E7 E7 E7 E7 E7 E7 E7

                 --^--
```

Disk Fixer
- can't read T22,S01, there's nothing there
- the rest of the disk seems normal
- T00 -> ProDOS bootloader and catalog

Why didn't COPYA / Locksmith FDB work?
- There is an intentionally corrupted sector on track $22, which is almost certainly being verified later by a runtime protection check.

EDD worked. What does that tell us?
- The runtime protection check probably just checks that the corrupt T22,S01 is unreadable (nothing fancy with timing bits or half/quarter tracks).

Next steps:

1. Use a sector editor to search for obvious signs of sector reads
2. If that fails, trace the first .SYSTEM file
3. I don't know, go feed the ducks or something?

Chapter 1
How Do I Read Thee? Let Me Count The Ways

On the theory that some code on disk is trying to access track $22, and thus noticing if it's unexpectedly readable, let's enumerate some of the ways that could happen:

- Reading a file that is mapped to the unreadable sector on track $22. Copy II+ "Disk Map" shows there are no files mapped to track $22, so let's rule that out.

- Manually seeking to the track and looking for a nibble sequence. Given that this disk is ProDOS-based, there is no explicit support for "seeking to a particular track" unless you're calling ProDOS internals. (But that's always possible, of course!) Without calling into ProDOS, this technique would require low-level disk access (turning on the drive and hitting the right stepper motors and whatnot). A sector search with Disk Fixer didn't find any suspicious instances of

  ```
  "BD 89 C0" (LDA $C089,X)   ; drive on
  "AD E9 C0" (LDA $C0E9)     ; drive on
  "BD 80 C0" (LDA $C080,X)   ; stepper
  ```

  or any similar variations that would point to low-level disk access.

- Issuing a ProDOS MLI "raw block read"
  and checking the return code. This is
  a popular technique under ProDOS,
  partly because it can be adapted to
  work on 3.5-inch and 5.25-inch disks.
  Combined with the knowledge that EDD
  bit copy produced a working copy, I
  suspect this is what I'm looking for.

A sector search for "20 00 BF 80" (JSR
$BF00 / $80 -- the standard way to call
the ProDOS MLI to read a block) turned
up nothing outside the PRODOS file,
which always has one. Which means,
perhaps, that this entire exercise was
just mental gymnastics.

Or was it? Dun dun DUN...

Turning to the disk itself, the catalog
is revealing... and heartbreaking.

```
[S7,D1=ProDOS hard drive]
[S6,D1=non-working copy]

]PR#7
...
```

```
]CAT,S6,D1

/ANCIENT.DSK1

 NAME                TYPE  BLOCKS  MODIFIED

 GAME.PART.1          BIN     111  20-JUL-16
 PLVM02.SYSTEM        SYS       7  20-JUL-16
 CMD                  SYS      12  20-JUL-16
 PRODOS               SYS      34   7-AUG-13

BLOCKS FREE:  109     BLOCKS USED:  171
```

And right away, my heart sinks into my chest as I remember watching the KansasFest presentation that introduced this game. One key fact leaps to mind: the entire game is written in PLASMA, a modern interpreted language for the Apple II by David Schmenk. <http://github.com/dschmenk/PLASMA>

PLVM02.SYSTEM stands for PLASMA VIRTUAL MACHINE.

GAME.PART.1.BIN is full of opcodes, but not 6502 opcodes. PLASMA bytecode.

CMD appears to be some middleware that sits between ProDOS and PLASMA. It contains the string "Welcome to LegendOS" (printed on screen during boot), so it's definitely app-specific. No copy protection routines, though.

Oh God. The copy protection is written in PLASMA.

On the bright side, there is ample documentation of PLASMA bytecode:
<http://github.com/dschmenk/PLASMA
 #the-bytecodes>

On the not-so-bright side, bytecode.

Now what?

Chapter 2
Hook, Line, and Stinker

I've cracked protection-in-bytecode
before. Some Davidson disks implemented
their bad block checks in Pascal; other
disks did it in Forth. My working
theory is that this copy protection is

   (a) a bad block check
   (b) via the ProDOS MLI
   (c) using MLI command $80
   (d) called from PLASMA bytecode.

Condition (a) is based on the fact that
my EDD bit copy booted successfully.
Conditions (b) through (d) are guesses
at this point, albeit educated ones.

Let's test conditions (b) and (c) by
setting up a ProDOS MLI hook.

```
]PREFIX /ANCIENT.DSK1
]CALL-151

; set up hook in ProDOS MLI entry point
0300-   A9 4C          LDA   #$4C
0302-   8D 00 BF       STA   $BF00
0305-   A9 10          LDA   #$10
0307-   8D 01 BF       STA   $BF01
030A-   A9 03          LDA   #$03
030C-   8D 02 BF       STA   $BF02
030F-   60             RTS

; hook is here --
; get a pointer to the top address on
; the stack
0310-   BA             TSX
0311-   BD 02 01       LDA   $0102,X
0314-   8D 32 03       STA   $0332
```

```
; print it
0317-   20 DA FD    JSR   $FDDA
031A-   BD 01 01    LDA   $0101,X

; also store it in upcoming instruction
031D-   8D 31 03    STA   $0331
0320-   20 DA FD    JSR   $FDDA

; print space
0323-   A9 A0       LDA   #$A0
0325-   20 F0 FD    JSR   $FDF0

; increment stack return address
0328-   EE 31 03    INC   $0331
032B-   D0 03       BNE   $0330
032D-   EE 32 03    INC   $0332

; now this will get the byte after the
; return address, which is the ProDOS
; MLI command code
0330-   AD FF FF    LDA   $FFFF

; print that
0333-   20 DA FD    JSR   $FDDA

; print <CR>
0336-   A9 8D       LDA   #$8D
0338-   20 F0 FD    JSR   $FDF0

; and jump to real ProDOS MLI entry
; point (not sensitive to accumulator
; or any flags, so don't worry about
; saving/restoring anything)
033B-   4C 4B BF    JMP   $BF4B

*BSAVE HOOK,A$300,L$3E
```

```
; install the hook
*300G

; run the game
*-PLVM02.SYSTEM

BE84 C4                 (GET_FILE_INFO)
BE84 CC                 (CLOSE)
BE84 C8                 (OPEN)
BE84 D1                 (GET_EOF)
BE84 CA                 (READ)
BE84 CC                 (CLOSE)

0101-    A=00 X=00 Y=92 P=B0 S=F6
*
```

Hmm.

Based on the stack return address, it appears that all of these MLI calls are internal to ProDOS. In other words, we just saw what always happens when you execute a program. But as soon as the PLVM02.SYSTEM file takes over, our hook crashes.

Further investigation reveals that the hook I set up at $BF00 (to jump to $0310) is still in place, and my hook code at $0310 has not been corrupted. The real ProDOS MLI (at $BF4B) is also the same as it always was. So what's causing the crash?

```
 ______________________________
{                              }
{ "How often have I said to    }
{  you that when you have      }
{  eliminated the impossible,  }
{  whatever remains, however   }
{  improbable, must be the     }
{  truth?"                     }
{                              }
{          The Sign of the Four }
{______________________________}
```

If my code at $0310 is the same as it was when I started, AND it's still being called correctly from $BF00, AND it's still calling the correct code afterwards at $BF4B... there's only one other thing that could have changed: the print routines at $FDDA and $FDF0.

Because they're being swapped out.

With RAM bank switching.

To test *that* theory, I get to rewrite my hook code so it doesn't use any ROM routines.

Chapter 3
But Wait, There's More!

I moved the start of my hook to $02F0, because it gives me a little more room in page 3. I don't know if this is necessary, but OK.

```
; set up MLI hook
02F0-   A9 4C       LDA   #$4C
02F2-   8D 00 BF    STA   $BF00
02F5-   A9 00       LDA   #$00
02F7-   8D 01 BF    STA   $BF01
02FA-   A9 03       LDA   #$03
02FC-   8D 02 BF    STA   $BF02
02FF-   60          RTS

; hook is here --
; get a pointer to the MLI code (after
; the return address on the stack)
0300-   BA          TSX
0301-   BD 02 01    LDA   $0102,X
0304-   8D 17 03    STA   $0317
0307-   BD 01 01    LDA   $0101,X
030A-   8D 16 03    STA   $0316
030D-   EE 16 03    INC   $0316
0310-   D0 03       BNE   $0315
0312-   EE 17 03    INC   $0317

; save MLI code in memory (at $0321+)
0315-   AD FF FF    LDA   $FFFF
0318-   8D 21 03    STA   $0321
031B-   EE 19 03    INC   $0319

; continue with real MLI
031E-   4C 4B BF    JMP   $BF4B

*BSAVE HOOK2,A$2F0,L$31

; wipe the buffer where we'll be saving
; MLI commands
*321:00 N 322<321.3CEM
```

```
; install the hook
*2F0G

; run the game
*-PLVM02.SYSTEM
...loads game, exits to prompt...

]CALL -151

*321.

0321- .. C4 CC C8 D1 CA CC C7
0328- CC C8 CA CC C7 CC C8 CA
0330- CA CE CA CE CA CA CA CE
0338- CA CE CA CA CA CA CE CA
0340- CA CE CA CE CA CE CA CE
0348- CA CC 80 00 00 00 00 00
            ^^
      raw block read!
```

This has been an unqualified success.
Things we've now learned or verified:

(1) RAM bank switching was the cause of the previous hook crashing. Removing the calls to ROM ($FDDA and $FDF0) allowed the game to load far enough to trigger the protection routine.

(2) There is one, and only one, block read, which strongly supports my theory that the protection is using the ProDOS MLI to read the bad block.

(3) MLI $80 is the final ProDOS MLI
    command issued before the game
    shuts down. This strongly
    supports my theory that the
    protection is contingent on the
    return value of this function.

(4) Not shown, but the rest of page 3
    that I had cleared with zeroes is
    still zeroes, which means the
    game does not use page 3 before
    (or during) the protection check.
    This gives me some breathing room
    to install more complicated hooks
    later on.

But wait, we can learn even more! My
hook code is self-modifying, storing
the address of the MLI code at $0316/7
in order to copy it to the buffer at
$0321+. Since MLI command $80 was the
last ProDOS command to be issued before
the game exited, I can find the code
that issed the fatal command.

```
*315L

0315-   AD 23 95    LDA   $9523
0318-   8D 4B 03    STA   $034B
031B-   EE 19 03    INC   $0319
031E-   4C 4B BF    JMP   $BF4B
```

The routine appears to start at $951B:

```
*951BL

951B-   A0 00       LDY   #$00
951D-   20 06 08    JSR   $0806
9520-   20 00 BF    JSR   $BF00
9523-   80          ???
9524-  [DE 94]
9526-   B0 02       BCS   $952A
9528-   A9 00       LDA   #$00
952A-   2C 83 C0    BIT   $C083
952D-   60          RTS

*94DE.

94DE- .. .. .. .. .. .. 03 60
94E0- 00 40 17 01
```

So we're reading slot 6, drive 1, block $117, into $4000. That's the bad block!

But this routine is generic. Searching
for the exact sequence "20 00 BF 80"
turned up nothing; that was the first
thing I tried in chapter 1. Ah! But
searching for "20 06 08 20 00 BF" does
find this code in its raw form:

```
                 --v--

T08,S02
----------- DISASSEMBLY MODE ----------
00EC:A0 00             LDY   #$00
00EE:20 06 08          JSR   $0806
00F1:20 00 BF          JSR   $BF00
00F4:00                BRK
00F5:00                BRK
00F6:00                BRK
00F7:B0 02             BCS   $00FB
00F9:A9 00             LDA   #$00
00FB:2C 83 C0          BIT   $C083
00FE:60                RTS

                 --^--
```

See? It's just a stub for MLI calls.
All the relevant details -- the MLI
command code and parameter table
address -- are filled in at runtime by
the caller.

It's time to pop the stack.

Chapter 4
In Which We Try To Resist The Urge To Make A "Pop Goes The Weasel" Joke

The next version of my ProDOS MLI hook
is the same as the previous version,
until here:

```
; get currently executing MLI command
0315-    AD FF FF     LDA   $FFFF

; is it the block read? (there's only
; one)
0318-    C9 80        CMP   #$80
031A-    F0 03        BEQ   $031F

; no, continue
031C-    4C 4B BF     JMP   $BF4B

; yes! save the stack
031F-    A0 00        LDY   #$00
0321-    B9 00 01     LDA   $0100,Y
0324-    99 00 21     STA   $2100,Y
0327-    C8           INY
0328-    D0 F7        BNE   $0321

; switch back to ROM
032A-    AD 82 C0     LDA   $C082

; break directly to the monitor
032D-    4C 59 FF     JMP   $FF59

*BSAVE HOOK3,A$2F0,L$40

; install the hook
*2F0G
```

```
; run the game
*-PLVM02.SYSTEM
...loads game, breaks to monitor...
```

Success! Let's see where the stack leads us.

```
*2100.
...
21C8- FF FF FF FF FF FF 2A 2A
21D0- B0 2A 62 22 95 82 09 0F
               ^^^^^ ^^^^^
                now   next

21D8- D7 D9 11 A8 1F 05 67 0F
21E0- D7 D9 2C 4C 20 02 69 0F
21E8- D7 D9 05 76 22 03 6C 0F
21F0- 25 DA 2D 28 1F 04 70 0F
21F8- D7 D9 08 98 1F 70 0F 00
```

The next return address on the stack is $0982, so after the ProDOS MLI call, we should return to $0983.

That routine appears to start at $093F:

```
*93FL

093F-   2C 83 C0    BIT   $C083
0942-   2C 83 C0    BIT   $C083
0945-   AD 98 BF    LDA   $BF98
0948-   29 C8       AND   #$C8
094A-   C9 88       CMP   #$88
094C-   D0 0A       BNE   $0958
094E-   A9 10       LDA   #$10
0950-   8D AA C0    STA   $C0AA
0953-   A9 0B       LDA   #$0B
0955-   8D AB C0    STA   $C0AB
0958-   E0 11       CPX   #$11
095A-   B0 33       BCS   $098F
095C-   88          DEY
095D-   84 02       STY   $02
095F-   68          PLA
0960-   A8          TAY
0961-   68          PLA
0962-   C8          INY
0963-   8C 81 09    STY   $0981      o_O
0966-   D0 02       BNE   $096A
0968-   69 01       ADC   #$01
096A-   8D 82 09    STA   $0982      o_O
096D-   8A          TXA
096E-   65 02       ADC   $02
0970-   48          PHA
0971-   C9 11       CMP   #$11
0973-   B0 1A       BCS   $098F
0975-   AD 70 0D    LDA   $0D70
0978-   C9 AA       CMP   #$AA
097A-   D0 2D       BNE   $09A9
[...]
```

```
097C-   B5 C0           LDA   $C0,X
097E-   B4 D0           LDY   $D0,X
0980-   20 20 95        JSR   $9520   <-- !
0983-   85 02           STA   $02
0985-   68              PLA
0986-   AA              TAX
0987-   A5 02           LDA   $02
0989-   95 C0           STA   $C0,X
098B-   98              TYA
098C-   95 D0           STA   $D0,X
098E-   60              RTS
```

As the stack suggested, we called $9520 from $0980 and would be returning to it if we hadn't so rudely interrupted the call stack. I suppose I shouldn't be surprised that the JSR itself is the result of self-modifying code. (The address was popped off the stack at $095F and $0961, then set at $0963 and $096A.)

Once we return from $9520, the routine pops one byte off the stack (at $0985) to restore the X register before returning, so the return address is $D9D7+1 = $D9D8.

That's not a valid entry point in ROM, but look! The $C083 softswitches at $093F and $0942 select RAM bank 2. So I assume the return address is in bank 2.

```
; copy F8 ROM to RAM bank 2
*C081 C081 F800<F800.FFFFM

; switch fully to RAM bank 2
*C083 C083
```

Because I copied the F8 ROM, monitor routines still work, so now I can poke around the code in RAM bank 2 in situ.

$D9D8 is part of a routine that appears to start at $D9B7:

```
*D9B7L

D9B7-   C8          INY
D9B8-   D0 02       BNE   $D9BC
D9BA-   E6 F5       INC   $F5
D9BC-   B1 F4       LDA   ($F4),Y
D9BE-   85 E5       STA   $E5
D9C0-   C8          INY
D9C1-   D0 02       BNE   $D9C5
D9C3-   E6 F5       INC   $F5
D9C5-   B1 F4       LDA   ($F4),Y
D9C7-   85 E6       STA   $E6
D9C9-   A5 F5       LDA   $F5
D9CB-   48          PHA
D9CC-   A5 F4       LDA   $F4
D9CE-   48          PHA
D9CF-   98          TYA
D9D0-   48          PHA
D9D1-   8D 02 C0    STA   $C002
D9D4-   58          CLI
D9D5-   20 39 DA    JSR   $DA39   <-- !
D9D8-   78          SEI
D9D9-   8D 03 C0    STA   $C003
D9DC-   68          PLA
D9DD-   A8          TAY
D9DE-   68          PLA
D9DF-   85 F4       STA   $F4
D9E1-   68          PLA
D9E2-   85 F5       STA   $F5
D9E4-   A9 D3       LDA   #$D3
D9E6-   85 FA       STA   $FA
D9E8-   4C F0 00    JMP   $00F0
```

If we're returning to $D9D8, we must
have just called the JSR $DA39 at
$D9D5.

```
*DA39L

DA39-   6C E5 00    JMP   ($00E5)

*E5.E6

00E5- 1B 95
```

...which is indeed where we were called
from: $951B.

Then we pop three more bytes off the
stack (at $D9DC, $D9DE, and $D9E1) to
restore Y and store the others in zero
page, then we jump to... zero page.

And now we're really getting somewhere.

```
*F0L

00F0-   C8          INY
00F1-   F0 08       BEQ   $00FB
00F3-   B9 A8 1F    LDA   $1FA8,Y
00F6-   85 F9       STA   $F9
00F8-   6C 54 D3    JMP   ($D354)
00FB-   E6 F5       INC   $F5
00FD-   D0 F4       BNE   $00F3
```

This appears to be the main loop of the
PLASMA interpreter, which is good,
because that means I can find out which
bytecode PLASMA is interpreting when it
issues the (ultimately fatal) MLI block
read command.

From my stack capture (still in memory
at $2100), I can see which values will
go into Y, $F4, and $F5.

```
*2100.
...
21C8- FF FF FF FF FF FF 2A 2A
21D0- B0 2A 62 22 95 82 09 0F
21D8- D7 D9 11 A8 1F 05 67 0F
            ^^ ^^^^^
            Y  ($F4)
```

So Y = #$11, and ($F4) -> $1FA8.

```
*1FA8.

1FA8- 00 00 00 00 00 00 00 00
1FB0- 00 00 00 00 00 00 00 00
1FB8- 00 00 00 00 00 00 00 00
1FC0- 00 00 00 00 00 00 00 00
1FC8- 00 00 00 00 00 00 00 00
1FD0- 00 00 00 00 00 00 00 00
1FD8- 00 00 00 00 00 00 00 00
1FE0- 00 00 00 00 00 00 00 00
1FE8- 00 00 00 00 00 00 00 00
1FF0- 00 00 00 00 00 00 00 00
1FF8- 00 00 00 00 00 00 00 00
```

Wait, what?

I'm no expert in PLASMA bytecode (*),
but I don't think that's right.

(*) not guaranteed, actual expertise
    may vary

Well, there is one other $1FA8... in auxiliary memory. This game requires 128K. Maybe auxmem is active when the PLASMA interpreter is reading bytecode?

A short program to test that theory:

```
; copy auxmem to main memory
; $0800..$BFFF
0300-   A9 00         LDA   #$00
0302-   85 3C         STA   $3C
0304-   85 42         STA   $42
0306-   A9 08         LDA   #$08
0308-   85 3D         STA   $3D
030A-   85 43         STA   $43
030C-   A9 FF         LDA   #$FF
030E-   85 3E         STA   $3E
0310-   A9 BF         LDA   #$BF
0312-   85 3F         STA   $3F
0314-   18            CLC
0315-   4C 11 C3      JMP   $C311

*300G

*1FA8.

1FA8- 58 05 02 26 23 95 66 00
1FB0- 70 26 24 95 66 02 72 54
1FB8- 1B 95 74 04 64 04 5A 58
1FC0- 05 02 66 00 66 02 54 88
```

Booyah.

Now I get to read bytecode in a language I don't understand.

According to the PLASMA documentation
<http://github.com/dschmenk/PLASMA>,
opcode #$54 is "sub routine call with
stack parameters." Like a "JSR", only
PLASMA-fied. Which means that

```
1FB0- .. .. .. .. .. .. .. 54
1FB8- 1B 95
```

...should call $951B. Which is exactly
what the interpreter had just done
before I so rudely interrupted it.

```
1FB8- .. .. 74 04
```

means "store top of stack into local
byte at frame offset." The #$04 is an
index into the local frame, which is
created for PLASMA functions to use.
The value on the top of the (PLASMA)
stack is the result code that was
returned from the ProDOS MLI call, way
back at $9520.

```
1FB8- .. .. .. .. 64 04
```

means "load byte from frame offset." So
we're loading the byte we just stored.

```
1FB8- .. .. .. .. .. .. 5A
```

means "deallocate frame and return from
sub routine call." Like an "RTS", only
PLASMA-fied.

Uh oh. We've reached the end of the current function, but the MLI return code isn't checked until we return to the caller. Where's the caller?

Pop goes the weasel. (*)

(*) Damn it.

Chapter 5
Hack Everything, We're Doing 5 Blades

To find out where to look next, we get to dig even further into the PLASMA interpreter. The main loop is on zero page, and it looks like this:

```
*F0L

00F0-   C8          INY
00F1-   F0 08       BEQ   $00FB
00F3-   B9 A8 1F    LDA   $1FA8,Y
00F6-   85 F9       STA   $F9
00F8-   6C 54 D3    JMP   ($D354)
00FB-   E6 F5       INC   $F5
00FD-   D0 F4       BNE   $00F3
```

$F4, $F5, and the Y register are set in advance, so the statement at $F3 changes but always gets the next opcode. Then we immediately store that opcode in... $F9, which is part of the following instruction. Oh! Every opcode is the low byte of the address of the handler for that opcode! The $D300 page is one giant jump table!

That means, when we execute the opcode #$5A, we end up jumping to ($D35A).

```
*D35A.D35B

D35A- 6A DA

*DA6AL

DA6A-   8D 02 C0    STA   $C002
DA6D-   58          CLI
DA6E-   68          PLA
DA6F-   18          CLC
DA70-   65 E0       ADC   $E0
DA72-   85 E2       STA   $E2
DA74-   A9 00       LDA   #$00
DA76-   65 E1       ADC   $E1
DA78-   85 E3       STA   $E3
DA7A-   68          PLA
DA7B-   85 E0       STA   $E0
DA7D-   68          PLA
DA7E-   85 E1       STA   $E1
DA80-   60          RTS
```

OK, we pop three bytes off the stack
(at $DA6E, $DA7A, and $DA7E), then RTS.

According to our captured stack dump:

```
21D8- D7 D9 11 A8 1F 05 67 0F
                     ^^ ^^ ^^
          goes to zp$E2/E0/E1

21E0- D7 D9 2C 4C 20 02 69 0F
      ^^^^^
 return address
```

And now we're back to $D9D8, the same
routine that led us to $00F0 earlier.

```
*D9D8L

D9D8-   78          SEI
D9D9-   8D 03 C0    STA   $C003
D9DC-   68          PLA
D9DD-   A8          TAY
D9DE-   68          PLA
D9DF-   85 F4       STA   $F4
D9E1-   68          PLA
D9E2-   85 F5       STA   $F5
D9E4-   A9 D3       LDA   #$D3
D9E6-   85 FA       STA   $FA
D9E8-   4C F0 00    JMP   $00F0
```

Three more values come off the stack and go into Y and $F4/F5. According to my stack dump:

```
21E0- D7 D9 2C 4C 20 02 69 0F
            ^^ ^^^^^
            Y  ($F4)
```

So the next opcode PLASMA interprets should be at ($F4) + Y + 1, which is $204C + $2C + 1, which is $2079.

```
*204C

204C-  .. .. .. .. 26 E4 94 2A
2050-  01 54 46 95 20 4C 0B 00
2058-  00 54 2E 95 30 54 60 16
2060-  30 00 54 2E 95 30 2C 50
2068-  C0 60 30 2C 17 01 7A E2
2070-  94 2A 80 26 DE 94 54 88
                         ^^^^^
2078-  94
       ^^
```

Promising: the opcodes immediately before $2079 are a subroutine call. That's the function we just returned from!

Now execution continues at $2079.

```
2078-  .. 76 00
```

means "store top of stack into local word at frame offset 0." We're storing the return value from the subroutine in this function's local frame. (That return value, in turn, was the return value from the ProDOS MLI call back at $9520.)

```
2078-  .. .. .. 66 00
```

means "load word from frame offset 0." We're loading the return value.

```
2078-  .. .. .. .. .. 00
```

means "push zero on the stack." Just what it says on the tin.

```
2078- .. .. .. .. .. .. 40
```

means "if next from top is equal to top, set top true." We're comparing the ProDOS MLI return value to 0.

```
2078- .. .. .. .. .. .. .. 4C
2080- 04
```

means "branch if top of stack is zero." We're branching based on whether the ProDOS MLI call returned 0 or not.

And finally, I see an opportunity to make a small patch. If I change the #$40 opcode to #$42 ("if next from top is NOT equal to top, set top true"), I can invert the bad block check so the game only loads if there is NOT a bad block on track $22.

Turning to my trusty Disk Fixer sector editor, I do a search for the hex sequence "76 00 66 00 00 40 4C 04" and find exactly one match on track $08.

```
T08,S0F,$98: 40 -> 42
```

```
]PR#6
...works, and it is glorious...
```

Quod erat liberandum.

Changelog

2016-09-09

- typo [thanks qkumba]

2016-09-08

- initial release