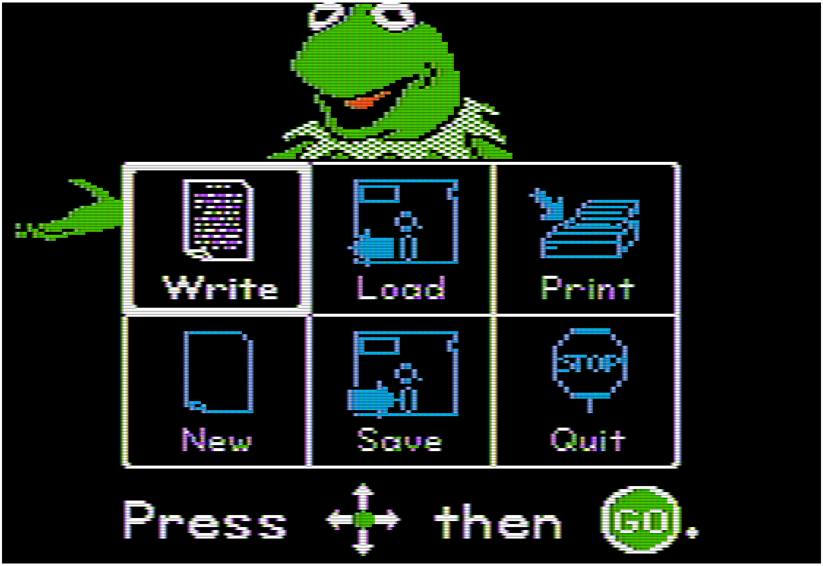
Muppet Slate



2017-04-24

4am to deprotect and preserve

Contents

Muppet Slate™
VERSION 1.0
Tad Wood
Scott Clough
Donna Stanger
© 1988 Henson Associates, Inc.

```
--------------Muppet Slate---------------
A 4am crack                   2017-04-24
----------------------------------------

Name: Muppet Slate
Version: 1.0
Genre: educational
Year: 1988
Authors: Tad Wood, Scott Clough, Donna
  Stanger
Publisher: Sunburst Communications
Platform: Apple ][+ or later
Media: 2 single-sided 5.25-inch disks
OS: ProDOS 1.4 QB
Previous cracks: none
Similar cracks:
  #113 1-2-3 Sequence Me
```

Chapter 0
In Which Various Automated Tools Fail In Interesting Ways

COPYA
  No errors, but the copy boots ProDOS,
  displays a company logo, then quickly
  switches to a graphical error message
  "Unable to load Muppet Slate. Press
  RETURN to reboot."

Locksmith Fast Disk Backup
  ditto

EDD 4 bit copy (no sync, no count)
  ditto

Copy][+ nibble editor
  nothing on track $23, no obvious
  oddities elsewhere

Disk Fixer
  T00,S00 looks like standard ProDOS
  bootloader. Track $00 has a disk
  catalog, and setting "DOS type" to
  "[P]RODOS" allows me to see a full
  directory listing and follow
  individual files. Nothing suspicious.

Why didn't any of my copies work?
  Probably a nibble check during boot.
  There's nothing wrong with the copy
  except that it's, you know, not an
  original. Something is going out of
  its way to display a pretty graphical
  error message to tell me to go f---
  myself.

Next steps:

1. Search for common markers of a nibble check
2. If that fails, trace the boot through the startup program
3. If that fails, I dunno, go feed the ducks or something?

Chapter 1
In Which We Get Lucky

Since my copy goes down a different code path than the original, I'm guessing there is a runtime protection check somewhere. One thing that all protection checks have in common is they need to turn on the drive motor by accessing a specific address in the $C0xx range. For slot 6, it's $C0E9, but to allow disks to boot from any slot, developers usually use code like this:

```
   LDX <slot number x 16>
   LDA $C089,X
```

There's nothing that says where the slot number has to be, although the disk controller ROM routine uses zero page $2B and lots of disks just reuse that. There's also nothing that says you have to use the X-register as the index, or that you must use the accumulator as the load register. But most RWTS code does, out of convention I suppose (or possibly fear of messing up such low-level code in subtle ways).

Also, since developers don't actually want people finding their protection-related code, they may try to encrypt it or obfuscate it to prevent people from finding it. But eventually, the code must exist and the code must run, and it must run on my machine, and I have the final say on what my machine does or does not do.

But sometimes you get lucky.

Turning to my trusty Disk Fixer sector
editor, I search the non-working copy
for "BD 89 C0", which is the opcode
sequence for "LDA $C089,X".

```
[Disk Fixer]
  ["F"ind]
    ["H"ex]
      ["BD 89 C0"]

                 --v--

------------- DISK SEARCH -------------

$00/$0E-$A7   $04/$0D-$5C   $05/$0A-$3C
$05/$0B-$AF   $11/$0B-$C1

                 --^--
```

T00,S0E is part of the bootloader.

T04,S0D appears to be part of ProDOS.

T05,S0A and T05,S0B are suspicious.
According to Copy II Plus "disk map,"
these two sectors are part of the file
MS.SYSTEM, which is coincidentally the
first .SYSTEM file on the disk and
would be executed first after ProDOS
loads.

```
                 --v--

DISK MAP                SLOT 6  DRIVE 1
/MUPPET.SLATE.A/MS.SYSTEM

   TRACK            1               2
   0123456789ABCDEF0123456789ABCDEF012

S0 .....**............................
EE .....**............................
CD .....**............................
TC .....**............................
OB .....**............................
RA .....**............................
 9 .....**............................
 8 .....**............................
 7 .....**............................
 6 .....**............................
 5 .....**............................
 4 .....**............................
 3 .....*.............................
 2 .....*.............................
 1 .....*.............................
 F .....*.............................

   USE ARROW KEYS TO MAP OTHER FILES

                 --^--
```

Rebooting to my hard drive in slot 7 and dropping to a BASIC prompt, I can load MS.SYSTEM into memory and start poking around.

Chapter 2
Who Needs Device Independence Anyway?

```
[S7,D1=my hard drive, /A4AMCRACK/]
[S6,D1=non-working copy]

]PR#7
...

; Copy II Plus showed the disk volume
; name as "MUPPET.SLATE.A"
]PREFIX /MUPPET.SLATE.A

; ProDOS always loads .SYSTEM files at
; address $2000
]BLOAD MS.SYSTEM,A$2000,TSYS

]CALL -151

*2000L

; immediately copies itself to higher
; memory
2000-   A2 1A       LDX   #$1A
2002-   A0 00       LDY   #$00
2004-   B9 20 20    LDA   $2020,Y
2007-   99 00 60    STA   $6000,Y
200A-   C8          INY
200B-   D0 F7       BNE   $2004
200D-   EE 06 20    INC   $2006
2010-   EE 09 20    INC   $2009
2013-   CA          DEX
2014-   D0 EE       BNE   $2004

; and we continue from there
2016-   4C 00 60    JMP   $6000

; replace "JMP $6000" with "RTS"
*2016:60
```

```
; execute memory copy
*2000G

*6000L

; copies some code to $0350 and sets
; the reset vector to point to it
; (not shown)
6000-   20 3C 61    JSR   $613C

; check machine ID (not shown)
6003-   20 CE 60    JSR   $60CE

; load and display Sunburst logo
; (not shown, but the subroutine is
; self-contained and I can call it
; directly from the monitor and it
; displays the logo then exits
; gracefully)
6006-   20 BE 61    JSR   $61BE

; standard ProDOS initialization
; (not shown)
6009-   20 B8 60    JSR   $60B8

; hmm
600C-   20 05 62    JSR   $6205

*6205L

; initialize counter
6205-   A9 03       LDA   #$03
6207-   8D 29 64    STA   $6429

; try something
620A-   20 16 62    JSR   $6216

; if carry is clear, branch ahead and
; exit to caller
620D-   90 06       BCC   $6215
```

```
; otherwise decrement the counter and
; try again
620F-   CE 29 64     DEC   $6429
6212-   D0 F6        BNE   $620A

; if counter reached 0, set carry and
; exit to caller
6214-   38           SEC
6215-   60           RTS
```

So whatever is happening at $6216, we get three chances for it to succeed. That's... somewhat suspicious.

```
*6216L

6216-   A9 00        LDA   #$00
6218-   8D 24 64     STA   $6424
621B-   A9 08        LDA   #$08
621D-   8D 25 64     STA   $6425
6220-   AD 30 BF     LDA   $BF30
6223-   8D 23 64     STA   $6423
6226-   29 7F        AND   #$7F
6228-   85 0C        STA   $0C
622A-   20 E1 63     JSR   $63E1
```

```
*63E1L

; munge the boot slot into $Cx form
; (so boot from slot 6 -> $C6)
63E1-   AA          TAX
63E2-   29 70       AND   #$70
63E4-   4A          LSR
63E5-   4A          LSR
63E6-   4A          LSR
63E7-   4A          LSR
63E8-   18          CLC
63E9-   69 C0       ADC   #$C0
63EB-   8D 19 64    STA   $6419
63EE-   A0 01       LDY   #$01
63F0-   A9 20       LDA   #$20
63F2-   20 17 64    JSR   $6417

*6417L

6417-   D9 00 C1    CMP   $C100,Y
641A-   60          RTS
```

OK, I just figured it out. This routine is checking for magic bytes in the card firmware to detect whether we booted from a 5.25-inch floppy drive. The magic bytes to check are

```
  $Cx01 = #$20
  $Cx03 = #$00
  $Cx05 = #$03
  $CxFF = #$00
```

We self-modified the instruction at $6417 based on the boot slot, so it's comparing against $C600,Y if we booted from slot 6, $C500,Y if we booted from slot 5, &c. Lots of protection checks only work if you boot from slot 6, so I appreciate the effort, kind of.

```
63F5-   D0 1D         BNE   $6414

; check $Cx03
63F7-   A0 03         LDY   #$03
63F9-   A9 00         LDA   #$00
63FB-   20 17 64      JSR   $6417
63FE-   D0 14         BNE   $6414

; check $Cx05
6400-   A0 05         LDY   #$05
6402-   A9 03         LDA   #$03
6404-   20 17 64      JSR   $6417
6407-   D0 0B         BNE   $6414

; check $CxFF
6409-   A0 FF         LDY   #$FF
640B-   A9 00         LDA   #$00
640D-   20 17 64      JSR   $6417
6410-   D0 02         BNE   $6414

; carry the clear bit if the boot slot
; was identified as a 5.25-inch floppy
; drive
6412-   18            CLC
6413-   60            RTS
6414-   8A            TXA

; set the carry bit otherwise
6415-   38            SEC
6416-   60            RTS
```

Continuing from $622D...

```
622D-   90 22        BCC   $6251
.
. [some code omitted, but basically,
.  booting from anything but a floppy
.  drive is always treated as an error]
.
6251-   4C 6E 62     JMP   $626E
```

And away we go.

Chapter 3
It's Not A Phase, Mom, This Is Who I Am

Via the unconditional jump from $6251, execution continues at $626E:

```
*626EL

; save registers
626E-   08          PHP
626F-   78          SEI
6270-   A9 03       LDA   #$03
6272-   8D 26 64    STA   $6426
6275-   A9 00       LDA   #$00
6277-   8D 27 64    STA   $6427
627A-   20 1B 64    JSR   $641B

*641BL

641B-   20 00 BF    JSR   $BF00
641E-  [80]       ; MLI command
641F-  [22 64]    ; MLI parameter table
6421-   60          RTS
```

Ah! A raw block read. Never a good sign (outside of the PRODOS system file). And if the MLI parameter table is at $6422, then we just set it to read block 3 at $6270.

Returning to the caller and continuing from $627D:

```
; try one thing (more on this in a
; moment)
627D-   20 1A 63    JSR   $631A
```

```
; if that works, return to caller
; (I'm assuming here that we're keeping
; with the convention of "C clear =
; success, C set = failure" that the
; other routines were using.)
6280-   90 08          BCC   $628A

; try another thing
6282-   20 8D 62       JSR   $628D

; if that works, return to caller
6285-   90 03          BCC   $628A

; otherwise restore registers and set
; carry for caller
6287-   28             PLP
6288-   38             SEC
6289-   60             RTS

; (success path, from $6280 or $6285)
; restore registers and clear carry for
; caller
628A-   28             PLP
628B-   18             CLC
628C-   60             RTS
```

So there are two different things, and either one of them can succeed, and that's enough. But if they both fail, then the caller gets the failure signal of the carry bit set.

Here's the first thing, at $631A:

```
; turn on the drive motor manually
; (this is what led me to this code in
; the first place)
631A-    A6 0C          LDX   $0C
631C-    BD 89 C0       LDA   $C089,X
631F-    BD 8E C0       LDA   $C08E,X

; munge the boot slot, e.g. $C6 -> $EC,
; presumably so we can access the data
; latch ($C0EC) in a slot-independent
; fashion
6322-    18             CLC
6323-    A5 0C          LDA   $0C
6325-    69 8C          ADC   #$8C
6327-    8D 3B 63       STA   $633B
632A-    8D A4 63       STA   $63A4

; initialize Death Counter
632D-    A0 F0          LDY   #$F0
632F-    8C 28 64       STY   $6428
6332-    C8             INY
6333-    D0 05          BNE   $633A
6335-    EE 28 64       INC   $6428

; when Death Counter hits 0, branch to
; failure path
6338-    F0 62          BEQ   $639C
```

```
; find address prologue "D5 AA 96"
633A-   AD EC C0      LDA   $C0EC
633D-   10 FB         BPL   $633A
633F-   C9 D5         CMP   #$D5
6341-   D0 EF         BNE   $6332
6343-   20 A3 63      JSR   $63A3
6346-   C9 AA         CMP   #$AA
6348-   D0 F5         BNE   $633F
634A-   20 A3 63      JSR   $63A3
634D-   C9 96         CMP   #$96
634F-   D0 EE         BNE   $633F

6351-   20 A3 63      JSR   $63A3

*63A3L

; read a nibble and return it
63A3-   AD EC C0      LDA   $C0EC
63A6-   10 FB         BPL   $63A3
63A8-   60            RTS
```

Ah, this was one of the instructions we self-modified earlier (at $632A).

Continuing from $6354...

```
; I guess we're ignoring that nibble we
; just read, because now we're reading
; another one (which we also ignore).
6354-   20 A3 63      JSR   $63A3
```

```
; The next 4 nibbles need to be $AA.
; Since we're inside an address field
; and just skipped 2 nibbles, that
; means we're looking for T00,S00.
6357-   A2 03           LDX   #$03
6359-   20 A3 63        JSR   $63A3
635C-   C9 AA           CMP   #$AA
635E-   D0 D2           BNE   $6332
6360-   CA              DEX
6361-   10 F6           BPL   $6359

; skip 2 more nibbles (address field
; checksum)
6363-   20 A3 63        JSR   $63A3
6366-   20 A3 63        JSR   $63A3

; next nibble needs to be $DE (standard
; first nibble of the address epilogue)
6369-   20 A3 63        JSR   $63A3
636C-   C9 DE           CMP   #$DE
636E-   D0 C2           BNE   $6332

; next nibble needs to be $AA (standard
; second nibble of epilogue)
6370-   A6 0C           LDX   $0C
6372-   BD 8C C0        LDA   $C08C,X
6375-   10 FB           BPL   $6372
6377-   C9 AA           CMP   #$AA
6379-   D0 B7           BNE   $6332
637B-   A0 02           LDY   #$02

; reset data latch (!)
637D-   BD 8D C0        LDA   $C08D,X

; burn some CPU cycles (7 to be exact)
6380-   48              PHA
6381-   68              PLA
```

```
; now look for a $BB nibble
6382-   BD 8C C0    LDA   $C08C,X
6385-   10 FB       BPL   $6382
6387-   C9 BB       CMP   #$BB
6389-   F0 06       BEQ   $6391

; must find that $BB nibble within the
; next 3 nibbles after the epilogue
638B-   88          DEY
638C-   10 F4       BPL   $6382

; otherwise jump back to increment the
; Death Counter
638E-   4C 35 63    JMP   $6335

; execution continues here (from the
; "BEQ" at $6389, after we find the $BB
; nibble) --
; next nibble must be $F9
6391-   BD 8C C0    LDA   $C08C,X
6394-   10 FB       BPL   $6391
6396-   C9 F9       CMP   #$F9
6398-   D0 F4       BNE   $638E

; success path falls through to here --
; clear carry on the way out
639A-   18          CLC

; there's a hidden "SEC" instruction
; here (opcode $38) which is called if
; the Death Counter hits 0 ($6338
; branches to $639C, in the middle of
; this instruction as listed)
639B-   24 38       BIT   $38
```

```
; either way, turn off the drive motor
; on the way out
639D-    A6 0C        LDX   $0C
639F-    BD 88 C0     LDA   $C088,X
63A2-    60           RTS
```

OK, what the hell is going on?

Chapter 4
The Truth Is In The Bits

Turning to my trusty Copy II Plus nibbl
editor, I can easily look at track 0 an
find the nibbles that this routine is
checking by searching for "AA AA AA":

```
                 --v--

   COPY ][ PLUS BIT COPY PROGRAM 8.4
(C) 1982-9 CENTRAL POINT SOFTWARE, INC.
---------------------------------------

TRACK: 00  START: 1800  LENGTH: 3DFF

26E8: FF+FF+FF+FF+FF+FF+FF+FF+  VIEW
26F0: FF+FF+FF+FF+FF+FF+FF+FF+
26F8: FF+FF+FF+FF+FF+FF+FF+FF+
2700: FF+FF+FF+FF+FF+FF+FF+FF+
2708: D5 AA 96 AA AB AA AA AA  <-270D
2710: AA AA AB DE AA EB+97+DF+
2718: FF E7 F9 FE FF+FF+D5 AA
2720: AD AC B6 ED F2 FF 9E B7  FIND:
2728: FB D9 F9 FD F6 F9 DA EF  AA AA AA

---------------------------------------

  A  TO ANALYZE DATA  ESC TO QUIT

  ?  FOR HELP SCREEN  /  CHANGE PARMS

  Q  FOR NEXT TRACK   SPACE TO RE-READ

                 --^--
```

Nibbles with an extra "0" bit (called a
"timing bit") after them are normally
displayed in inverse. For the purposes
of plain text, I've reformulated them
as "+" signs instead.

Taking it nibble by nibble, starting at offset $2708, we

1. verify the address field prologue ($D5 $AA $96)
2. skip the disk volume number (2 nibbles)
3. verify the track + sector number ($AA $AA $AA $AA)
4. skip the address field checksum (2 nibbles)
5. verify the address field epilogue ($DE $AA)

That brings us to offset $2715, where I see $EB, $97, $DF, $FF, $E7, $F9...

But no $BB.

What would that nibble sequence look like on disk? The answer is, "It depends." Here is the simplest possible answer:

```
   |--EB--||--97--||--DF--||--FF--|
   11101011100101111101111111111111
```

But wait. Every nibble read from disk must have its high bit set. In theory, you could insert one or two "0" bits after any of those nibbles. (Two is the maximum, due to hardware limitations.) These "timing bits" would be swallowed by the standard "wait for data latch to have its high bit set" loop, which you see over and over in any RWTS code:

```
   :1    LDA $C08C,X
         BPL :1
```

And in fact, Copy II Plus is smart
enough to notice that some of those
nibbles had a timing bit after them.
So maybe the bitstream looks like this:

```
  |--EB--| |--97--| |--DF--| |--FF--|
  11101011010010111011011111011111111
          ^        ^        ^
       (extra)  (extra)  (extra)
```

$EB is followed by 1 timing bit,
$97 is followed by 1 timing bit, and
$DF is followed by 1 timing bit.

Totally legal, works on any Apple II
computer and any floppy drive. A normal
"LDA $C08C,X; BPL" loop would still
interpret this bitstream as "EB 97 DF".
Each of the extra "0" bits appear just
after we've just read a nibble and
we're waiting for the high bit to be
set again. So they get "swallowed."
Ignored. Like they were never there.

But what if we throw away the first few
bits of this bitstream, then restart
the bit-to-nibble interpretation? We
can do this by resetting the data latch
(LDA $C08D,X), which throws away any
bits that have been read up to that
point and starts over at 0.

During the time between reading the $AA
nibble and resetting the data latch,
the disk is still spinning and bits are
accumulating in the data latch. But as
soon as we reset the data latch, those
bits are lost.

How many bits? Let's count cycles.

```
6372-   BD 8C C0    LDA   $C08C,X
6375-   10 FB       BPL   $6372       ; 2
6377-   C9 AA       CMP   #$AA        ; 2
6379-   D0 B7       BNE   $6332       ; 2
637B-   A0 02       LDY   #$02        ; 2
637D-   BD 8D C0    LDA   $C08D,X
```

Each bit takes 4 CPU cycles to go by as the disk spins. After 8 cycles, we've missed the first 2 bits of the $EB nibble (marked below with an X):

```
            (normal start)

  |--EB--| |--97--| |--DF--| |--FF--|
  11101011010010111011011111011111111
  XX|--AD--|  |--BB--| |--FB--|

           (delayed start)
```

After resetting the data latch, the stream is interpreted as a completely different nibble sequence. Some of those "extra" bits are no longer being ignored. The timing bit after $EB is now being interpreted as data, as part of the $AD nibble. Other bits (inside $97) now occur in between the $AD and $BB nibbles, so they get "swallowed." One of the bits that was part of the $DF nibble is now being swallowed as well.

And there's our $BB nibble.

But this still isn't enough. The nibble after it is $FB, not $F9, so the protection check would still fail. But! Remember that each nibble can have up to two extra "0" bits after it. What if the original disk had two timing bits after the $DF nibble, instead of one?

```
            (normal start)
                             +--- here
                             v
  |--EB--| |--97--| |--DF--|  |--FF--|
  111010110100101110110111110011111111
  XX|--AD--|  |--BB--| |--F9--|

           (delayed start)
```

Aha! Now the desynchronized nibble stream goes $AD... $BB... $F9!

And there's the $F9 nibble we needed.

So the protection check only passes if
$EB is followed by 1 timing bit,
$97 is followed by 1 timing bit, and
$DF is followed by 2 timing bits.

Here's the kicker: even the best bit copiers couldn't tell the difference between 1 timing bit and 2 timing bits. There just isn't enough time to do it at 1 MHz while the disk is spinning independently of the CPU. By wasting just the right number of CPU cycles at just the right point in a specially crafted bitstream, developers could determine at runtime whether you had an original disk. And even the best bit copiers couldn't fool it.

Chapter 5
In Which We Get A Second Chance And Still Fail

Backing up to $626E, you may recall we were in a loop. If this routine at $631A failed, we called another routine at $628D. And if that failed, we tried each of them again, up to three times, before finally setting the carry (to indicate failure) and returning to the caller.

It turns out that $628D is another nibble desynchronization routine, but with slightly different timing after we reset the data latch. Here is the only substantial difference:

```
62EE-   A0 02       LDY   #$02
62F0-   BD 8D C0    LDA   $C08D,X
62F3-   BD 8C C0    LDA   $C08C,X
62F6-   10 FB       BPL   $62F3
62F8-   C9 BB       CMP   #$BB
62FA-   F0 06       BEQ   $6302
62FC-   88          DEY
62FD-   10 F4       BPL   $62F3
62FF-   4C A8 62    JMP   $62A8
```

The timing of the desynchronization is slightly tighter. After resetting the data latch (at $62F0), it immediately checks for the $BB nibble, instead of burning an additional 7 CPU cycles on PHA + PLA.

Needless to say, a copy is not going to pass this check either. It still relies on desynchronizing the bitstream by resetting the data latch, throwing away two bits of the $EB nibble and interpreting the remaining bitstream. It still requires the same 2 timing bits after the $DF nibble, and still, the most advanced bit copier would not be able to reproduce the bitstream.

Backing all the way up to $6205, I want this entire routine to clear the carry and return gracefully. No checking if we booted from a floppy drive. No fancy nibble desynchronization. No try-three-times-before-dying. Just CLC and RTS.

```
T05,S04,$25: A903 -> 1860

]PR#6
```

...works, and it is glorious...

As an added bonus, since we removed the floppy drive check, you can now copy all these files to a ProDOS hard drive and run the program from there.

Quod erat liberandum.

Acknowledgments

Thanks to qkumba for reviewing drafts of this write-up.