4 Master Boot Record Nibbles; or, One Boot Sector PoC Deserves Another

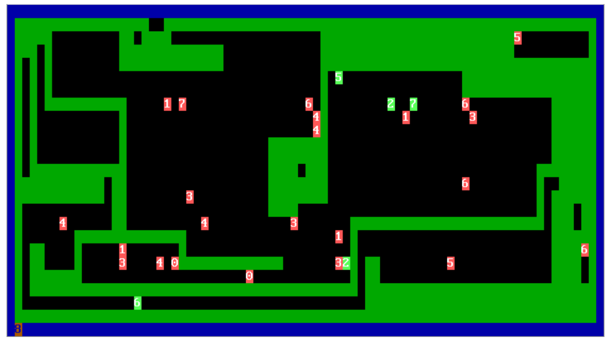
by Eric Davisson

I was inspired by the boot sector Tetranglix game by Juhani Haverinen, Owen Shepherd, and Shikhin Sethi published as PoC‖GTFO 3:8. I feel more creative when dealing with extreme limitations, and 512 bytes (510 with the `0x55AA` signature) of real-mode assembly sounded like a great way to learn BIOS API stuff. I mostly learned some `int 0x10` and `0x16` from this exercise, with a bit of `int 0x19` from a pull request.

The game looks a lot more like snake or nibbles, except that the tail never follows the head, so the game piece acts less like a snake and more like a streak left in Tron. I called it Tron Solitaire because there is only one player. This game has an advanced/dynamic scoring system with bonus and trap items, and progressively increasing game speed. This game can also be won.

I've done plenty of protected mode assembly and machine code hacking, but for some reason have never jumped down to real mode. Tetranglix gave me a hefty head start by showing me how to do things like quickly setting up a stack and some video memory. I would have possibly struggled a little with `int 0x16` keyboard handling without this code as a reference. Also, I re-used the elegant random value implementation as well. Finally, the PIT (Programmable Interval Timer) delay loop used in Tetranglix gave me a good start on my own dynamically timed delay.

I also learned how incredibly easy it was to get started with 16-bit real mode programming. I owe a lot of this to the immediate gratification from utilities like `qemu`. Looking at OS guides like the `osdev.org` wiki was a bit intimidating, because writing an OS is not at all trivial, but I wanted to start with much less than that. Just because I want to write real mode boot sector code doesn't mean I'm trying to actually boot something. So a lot of the instructions and guides I found had a lot of information that wasn't applicable to my unusual needs and desires.



I found that there were only two small things I needed to do in order to write this code: make sure the boot image file is exactly 512 bytes and make sure the last two bytes are `0x55AA`. That's it! All the rest of the code is all yours. You could literally start a file with `0xEBFE` (two-byte unconditional infinite "jump to self" loop), have 508 bytes of nulls (or ANYTHING else), and end with `0x55AA`, and you'll have a valid "boot" image that doesn't error or crash. So I started with that simple PoC and built my way up to a game.

The most dramatic space savers were also the least interesting. Instead of cool low level hacks, it usually comes down to replacing a bad algorithm. One example is that the game screen has a nice blue border. Initially, I drew the top and bottom lines, and then the right and left lines. I even thought I was clever by drawing the right and left lines together, two pixels at a time—because drawing a right pixel and incrementing brings me to the left and one row down. I used this side-effect to save code, rewriting a single routine to be both right and left.

However, all of this was still too much code. I tried something simpler: first splashing the whole screen with blue, then filling in a black box to only leave the blue border. The black box code still wasn't trivial, but much less code than the previous method. This saved me sixteen precious bytes!

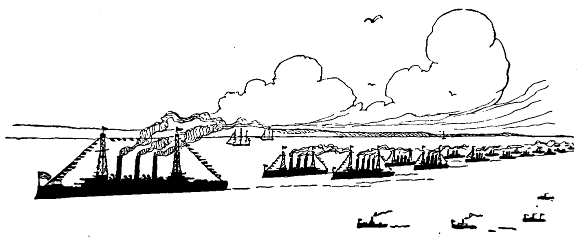
Less than a week after I put this on Github, my friend Darkvoxels made a pull request to change the game-over screen. Instead of splashing the screen red and idling, he just restarts the game. I liked this idea and merged. As his game-over is just a simple `int 0x19`, he saved ten bytes.

Although I may not have tons of reusable subroutines, I still avoided inlining as much as possible. In my experience, inlining is great for runtime performance because it cuts out the overhead of jumping around the code space and stack overhead. However, this tends to create more code as the tradeoff. With 510 effective bytes to work with, I would gladly trade speed for space. If I see a few consecutive instructions that repeat, I try to make a routine of it.

I also took a few opportunities to use self-modifying code to save on space. No longer do I have to manually hex hack the `w` bit in the `rwx` attribute in the `.text` section of an ELF header; real mode trusts me to do all of the "bad" things that dev hipsters rage at me about. So the rest of this article will be about these hacks.

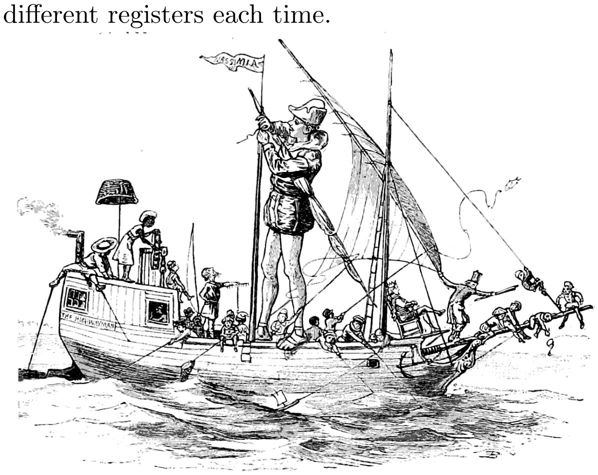
Two of the self-modifying code hacks in this code are similar in concept. There are a couple of places where I needed something similar to a global variable. I could push and pop it to and from the stack when needed, but that requires more bytes of code overhead than I had to spare. I could also use a dedicated register, but there are too few of those. On the other hand, assuming I'm actually using this dynamic data, it's going to end up being part of an operand in the machine code, which is what I would consider its persisted location. (Not a register, not the stack, but inside the actual code.)

As the pixel streak moves around on the gameboard, the player gets one point per character movement. When the player collects a bonus item of any value, this one-point-per gets three added to it, becoming a four-points-per. If another additional bonus item is collected, it would be up to 7 points. The code to add one point is `selfmodify: add ax, 1`. When a bonus item is collected, the routine for doing bonus points also has this line `add byte [selfmodify + 2], 3`. The `+2` offset to our `add ax, 1` instruction is the byte where the `1` operand was located, allowing us to directly modify it.



On a less technical note, this adds to the strategy of the game; it discourages just filling the screen up with the streak while avoiding items (so as to not create a mess) and just waiting out the clock. In fact, it is nearly impossible to win this way. To win, it is a better strategy to get as many bonuses as early as possible to take advantage of this progressive scoring system.

Another self-modifying code trick is used on the "win" screen. The background to the "YOU WIN!" screen does some color and character cycling, which is really just an increment. It is initialized with `winbg: mov ax, 0`, and we can later increment through it with `inc word [winbg + 0x01]`. What I also find interesting about this is that we can't do a space saving hack like just changing `mov ax, 0` to `xor ax, ax`. Yes, the result is the same; `ax` will equal `0x0000` and the `xor` takes less code space. However, the machine code for `xor ax, ax` is `0x31c0`, where `0x31` is the `xor` and `0xc0` represents "ax with ax." The increment instruction would be incrementing the `0xc0` byte, and the first byte of the next instruction since the `word` modifier was used (which is even worse). This would not increment an immediate value, instead it would do another `xor` of different registers each time.



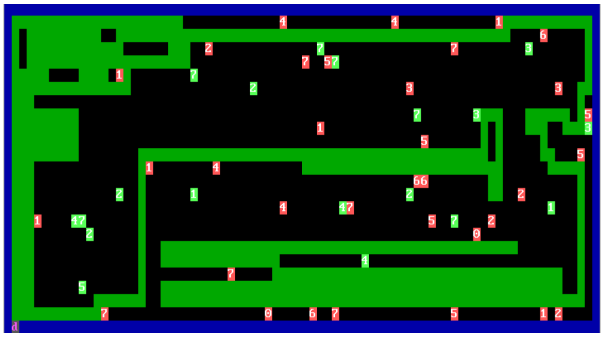
Also, instead of using an elaborate string print function, I have a loop to print a character at a pointer where my "YOU WIN!" string is stored (`winloop: mov al, [winmessage]`), and then use self-modifying code to increment the pointer on each round. (`inc byte [winloop + 0x01]`)

The most interesting self-modifying code in this game changes the opcode, rather than an operand. Though the code for the trap items and the bonus items have a lot of differences, there are a significant amount of consecutive instructions that are exactly the same, with the exception of the addition (bonus) or the subtraction (trap) of the score. This is because the score actually persists in video memory, and there is some code overhead to extract it and push it back before and after adding or subtracting to it.

So I made all of this a subroutine. In my assembly source you will see it as an addition (`math: add ax, cx`), even though the instruction initialized there could be arbitrary. Fortunately for me, the machine code format for this addition and subtraction instruction are the same. This means we can dynamically drop in whichever opcode we want to use for our current need on the fly. Specifically, the `add` I use is `ADD r/m16, r16` (`0x01` /r) and the `sub` I use is `SUB r/m16, r16` (`0x29` /r). So if it's a bonus item, we'll self modify the routine to add (`mov byte [math], 0x01`) and call it, then do other bonus related instructions after the return. If it's a trap item, we'll self modify the routine to subtract (`mov byte [math], 0x29`) and call it, then do trap/penalty instructions after the return. This whole hack isn't without some overhead; the most exciting thing is that this hack saved me one byte, but even a single byte is a lot when making a program this small!

I hope these tricks are handy for you when writing your own 512-byte game, and also that you'll share your game with the rest of us. Complete code and prebuilt binaries are available in the ZIP portion of this release.[8]

[8] `unzip pocorgtfo11.pdf tronsolitare.zip`



```
;Tron Solitare
;  *This is a PoC boot sector ( <512 bytes) game
;  *Controls to move are just up/down/left/right
;  *Avoid touching yourself, blue border, and the
;unlucky red 7

[ORG 0x7c00]         ;add to offsets
LEFT  EQU 75
RIGHT EQU 77
UP    EQU 72
DOWN  EQU 80

;Init the environment
;  init data segment
;  init stack segment allocate area of mem
;  init E/video segment and allocate area of mem
;  Set to 0x03/80x25 text mode
;  Hide the cursor
    xor ax, ax       ;make it zero
    mov ds, ax       ;DS=0

    mov ss, ax       ;stack starts at 0
    mov sp, 0x9c00   ;200h past code start

    mov ax, 0xb800   ;text video memory
    mov es, ax       ;ES=0xB800

    mov al, 0x03
    xor ah, ah
    int 0x10

    mov al, 0x03     ;Some BIOS crash without this
    mov ch, 0x26
    inc ah
    int 0x10

;Draw Border
    ;Fill in all blue
    xor di, di
    mov cx, 0x07d0   ;whole screens worth
    mov ax, 0x1f20   ;empty blue background
    rep stosw        ;push it to video memory

    ;fill in all black except for remaining blue edges
    mov di, 158      ;Almost 2nd row 2nd column (need
                     ;to add 4)
    mov ax, 0x0020   ;space char on black on black
    fillin:
    add di, 4        ;Adjust for next line and column
    mov cx, 78       ;inner 78 columns (exclude side
                     ;borders)
    rep stosw        ;push to video memory
    cmp di, 0x0efe   ;Is it the last col of last line
                     ;we want?
    jne fillin       ;If not, loop to next line

    ;init the score
    mov di, 0x0f02
    mov ax, 0x0100   ;#CHEAT (You can set the initial
                     ;score higher than this)
    stosw

    ;Place the game piece in starting position
    mov di, 0x07d0   ;starting position
    mov ax, 0x2f20   ;char to display
    stosw

mainloop:
    call random      ;Maybe place an item on screen

    ;Wait Loop
    ;Get speed (based on game/score progress)
        push di
        mov di, 0x0f02  ;set coordinate
        mov ax, [es:di] ;read data at coordinate
        pop di
        and ax, 0xf000  ;get most significant nibble
        shr ax, 14      ;now value 0-3
        mov bx, 4       ;#CHEAT, default is 4; make
                        ;amount higher for overall
                        ;slower (but still
                        ;progressive) game
        sub bx, ax      ;bx = 4 - (0-3)
        mov ax, bx      ;get it into ax

    mov bx, [0x046C];Get timer state
    add bx, ax      ;Wait 1-4 ticks (progressive
                    ;difficulty)
    ;add bx, 8      ;unprogressively slow cheat
    ;#CHEAT (comment above line out and uncomment
    ;this line)
    delay:
        cmp [0x046C], bx
        jne delay

    ;Get keyboard state
    mov ah, 1
    int 0x16
    jz persisted    ;if no keypress, jump to
                    ;persisting move state

    ;Clear Keyboard buffer
    xor ah, ah
    int 0x16

    ;Check for directional pushes and take action
    cmp ah, LEFT
    je left
    cmp ah, RIGHT
    je right
    cmp ah, UP
    je up
    cmp ah, DOWN
    je down
    jmp mainloop

    ;Otherwise, move in direction last chosen
    persisted:
    cmp cx, LEFT
    je left
    cmp cx, RIGHT
    je right
    cmp cx, UP
    je up
    cmp cx, DOWN
    je down

    ;This will only happen before first keypress
    jmp mainloop

    left:
        mov cx, LEFT    ;for persistenc
        sub di, 4       ;coordinate offset correction
        call movement_overhead
        jmp mainloop
    right:
        mov cx, RIGHT
        call movement_overhead
        jmp mainloop
    up:
        mov cx, UP
        sub di, 162
        call movement_overhead
        jmp mainloop
    down:
        mov cx, DOWN
        add di, 158
        call movement_overhead
        jmp mainloop

    movement_overhead:
        call collision_check
        mov ax, 0x2f20
        stosw
        call score
        ret

    collision_check:
        mov bx, di      ;current location on screen
        mov ax, [es:bx] ;grab video buffer + current
                        ;location

        ;Did we Lose?
            ;#CHEAT: comment out all 4 of these checks
            ;(8 instructions) to be invincible
            cmp ax, 0x2f20  ;did we land on green
                            ;(self)?
            je gameover
            cmp ax, 0x1f20  ;did we land on blue
                            ;(border)?
            je gameover
            cmp bx, 0x0f02  ;did we land in score
                            ;coordinate?
            je gameover
            cmp ax, 0xcf37  ;magic red 7
            je gameover

        ;Score Changes
            push ax         ;save copy of ax/item
            and ax, 0xf000  ;mask background
            cmp ax, 0xa000  ;add to score
            je bonus
            cmp ax, 0xc000  ;subtract from score
```

```
            je penalty
            pop ax          ; restore ax
        ret

    bonus:
        mov byte [math], 0x01
                            ;make itemstuff: routine use
                            ;add opcode
        call itemstuff
        stosw               ;put data back in
        mov di, bx          ;restore coordinate
        add byte [selfmodify + 2], 3

        ret
    penalty:
        mov byte [math], 0x29
                            ;make itemstuff: routine use
                            ;sub opcode
        call itemstuff
        cmp ax, 0xe000      ;sanity check for integer
                            ;underflow
        ja underflow
        stosw               ;put data back in
        mov di, bx          ;restore coordinate
        ret

        underflow:
            mov ax, 0x0100
            stosw
            mov di, bx
            ret

    itemstuff:
        pop dx              ;store return
        pop ax
        and ax, 0x000f
        inc ax              ;1-8 instead of 0-7
        shl ax, 8           ;multiply value by 256
        push ax             ;store the value

        mov bx, di          ;save coordinate
        mov di, 0x0f02      ;set coordinate
        mov ax, [es:di]     ;read data at coordinate and
                            ;subtract from score
        pop cx
        math:
        add ax, cx          ;'add' is just a suggestion...
        push dx             ;restore return
        ret

score:
    push di
    mov di, 0x0f02          ;set coordinate
    mov ax, [es:di]         ;read data at coordinate
    ;for each mov of character, add 'n' to score
        ;this source shows add ax, 1, however, each
        ;bonus item that is picked up increments this
        ;value by 3 each time an item is picked up.
        ;Yes, this is self modifying code, which is
        ;why the lable 'selfmodify:' is seen above, to
        ;be conveniently used as an address to pivot
        ;off of in an add byte [selfmodify + offset to
        ;'1'], 3 instruction
    selfmodify: add ax, 1 ;increment character in
                          ;coordinate
    stosw                   ;put data back in
    pop di
    ;Why 0xf600 as score ceiling:
        ;if it was something like 0xffff, a score from
        ;0xfffe would likley integer overflow to a low
        ;range (due to the progressive) scoring.
        ;0xf600 gives a good amount of slack for this.
        ;However, it's still "technically" possible to
        ;overflow; for example, hitting a '7' bonus
        ;item after already getting more than 171
        ;bonus items (2048 points for bonus, 514
        ;points per move) would make the score go from
        ;0xf5ff to 0x0001.
    cmp ax, 0xf600          ;is the score high enough to
                            ;'win' ;#CHEAT
    ja win
    ret

random:
    ;Decide whether to place bonus/trap
        rdtsc
        and ax, 0x000f
        cmp ax, 0x0007
        jne undo

    push cx                 ;save cx

    ;Getting random pixel
        redo:
        rdtsc               ;random
        xor ax, dx          ;xor it up a little
        xor dx, dx          ;clear dx
        add ax, [0x046C]    ;moar randomness
        mov cx, 0x07d0      ;Amount of pixels on screen
        div cx              ;dx now has random val
        shl dx, 1           ;adjust for 'even' pixel values

        ;Are we clobbering other data?
            cmp dx, 0x0f02  ;Is the pixel the score?
            je redo         ;Get a different value

            push di         ;store coord
            mov di, dx
            mov ax, [es:di] ;read data at coordinate
            pop di          ;restore coord
            cmp ax, 0x2f20  ;Are we on the snake?
            je redo
            cmp ax, 0x1f20  ;Are we on the border?
            je redo

    ;Display random pixel
        push di             ;save current coordinate
        mov di, dx          ;put rand coord in current

        ;Decide on item-type and value
        powerup:
        rdtsc               ;random
        and ax, 0x0007      ;get random 8 values
        mov cx, ax          ;cx has rand value
        add cx, 0x5f30      ;baseline
        rdtsc               ;random
        ;background either 'A' or 'C' (light green or
        ;red)
            and ax, 0x2000  ;keep bit 13
            add ax, 0x5000  ;turn bit 14 and 12 on
        add ax, cx          ;item-type + value

        stosw               ;display it
        pop di              ;restore coordinate

    pop cx                  ;restore cx

    undo:
    ret

gameover:
    int 0x19                ;Reboot the system and restart
                            ;the game.

    ;Legacy gameover, doesn't reboot, just ends with
    ;red screen
    ;xor di, di
    ;mov cx, 80*25
    ;mov ax, 0x4f20
    ;rep stosw
    ;jmp gameover

win:
    ;clear screen

    mov bx, [0x046C]        ;Get timer state
    add bx, 2
    delay2:
        cmp [0x046C], bx
        jne delay2

    mov di, 0
    mov cx, 0x07d0          ;enough for full screen
    winbg: mov ax, 0x0100
                            ;xor ax, ax wont work, needs to
                            ;be this machine-code format
    rep stosw               ;commit to video memory

    mov di, 0x07c4          ;coord to start 'YOU WIN!' message
    xor cl, cl              ;clear counter register
    winloop: mov al, [winmessage]
                            ;get win message pointer
    mov ah, 0x0f            ;white text on black background
    stosw                   ;commit char to video memory
    inc byte [winloop + 0x01]
                            ;next character
    cmp di, 0x07e0          ;is it the last character?
    jne winloop
    inc word [winbg + 0x01]
                            ;incrememnt fill char/fg/bg
                            ;(whichever is next)
    sub byte [winloop + 0x01], 14
                            ;back to first character upon
                            ;next full loop
    jmp win

    winmessage:
    db 0x02, 0x20
    dq 0x214e495720554f59   ;YOU WIN!
    db 0x21, 0x21, 0x20, 0x02

    ;BIOS sig and padding
    times 510-($-$$) db 0
    dw 0xAA55

    ; Pad to floppy disk.
    ;times (1440 * 1024) - ($ - $$) db 0
```