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 subrou-
tines, 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
...and ax push dx...  
...call 69 65...  
; Place the game piece in starting position  
55 jne cmp dx 49 add dx...  
...mov ax 45 43 mov ax 37 35...  
...init stack segment allocate area of mem  
; i n i t i a l environment  
DO DOWN  
9 RIGHT  
7 [Tron Solitaire]  
...; Score Changes  
mov bx, di...  
mov bx, [es:bx]...  
...; Clear Keyboard buffer  
xor ah, ah...  
...; Check for directional pushes and take action  
cmp ah, LEFT je sub dx...  
...; Otherwise, move in direction last chosen persisted  
cmp cx, LEFT je left...  
...; This will only happen before first keypress  
jmp mainloop  
left: mov cx, LEFT...  
...; Gameover  
cmp bx, 0x2f20...  
...; magic red 7  
je gameover  
cmp bx, 0xcf37...  
...; Save copy of ax/item...  
and ax, 0x0000...  
...; Subtract from score  
cmp ax, 0x0000...
je penalty
pop ax
; restore ax
ret

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

penalty:
mov byte [math], 0x29
; make itemstuff: routine use
sub ax
call itemstuff
cmp ax, 0x0000
; sanity check for integer
ja s: underflow
mov bx, bx: restore coordinate
mov ax, 0x0202
; set coordinate
mov ax, [0x046C]
; get data at coordinate
sub byte
; subtract from score
ja penalty
push dx
; restore return
ret

score:
ja score
mov ax, 0x0002
; restore return
mov ax, [0x0002]
; read data at coordinate
mov ax, [es:di]
; read data at coordinate and
sub byte
; subtract from score
ja score
pop cx
; restore cx
ret

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

push cx
; save cx

; Getting random pixel
redo:
rdtsc
xor dx, dx
; zero it up a little
xor ax, [0x046C]
; clear axe
add ax, [0x0046C]
; make randomness
mov cx, 0x07d0
; Amount of pixels on screen
div cx
; divide now has random val
shl ax, 1
; adjust for 'even' pixel values

; Are we clobbering other data?
cmp dx, 0x0002
; 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, 0x12f0
; 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
rand
add ax, 0x0007
; get random 8 values
mov cx, ax
; cx has rand value
adc cx, 0x5130
; baseline
rdtsc
; random
; background either 'A' or 'Q' (light green or red)
and ax, 0x2f20
; keep bit 7
add ax, 0x5000
; turn bit 14 and 12 on
add ax, cx
; item-type + value
stow
; display it
pop di
; restore coordinate
pop cx
; restore cx
ret

undo:

; Legacy gameover, doesn't reboot, just ends with:
; 'red screen
; zero di, dx
mov ax, 0xA2
mov bx, 0x00
rep stow
; commit to video memory

; win:
; clear screen
push bx
; [0xA46C]: Get timer state
add bx, 2
delay2:
cmp bx, 0x0A6C
bx
jne delay2
mov di, 0
mov dx, 0x07d0
; enough for full screen
winbg:
mov ax, 0x0010
; to az, az went work, needs to
; be this machine code format
rep stow
; commit to video memory
mov di, 0x07c4
; coord to start 'YOU WIN!' message
xor bx, cl
; clear counter register
winloop: mov al, [winmessage]
; get win message pointer
mov ah, 0x0f
; white text on black background
stow
; commit char to video memory
inc byte [winloop + 0x01]
; next char
jmp winloop
inc word [winbg + 0x01]
; increment fill char/fg/bg
; ( whichever is next)
sub byte [winloop + 0x01]
; back to first character upon
; next full loop
jmp win

winmessage:
db 0x02, 0x20

dq 0x21495720554f55
; 'YOU WIN!'
mov al, [winmessage]
; get win message pointer
mov ah, 0x0f
; white text on black background
stow
; commit char to video memory
inc byte [winloop + 0x01]
; next character
jne winloop
inc word [winbg + 0x01]
; increment fill char/fg/bg
; ( whichever is next)
sub byte [winloop + 0x01]
; back to first character upon
; next full loop
jmp win

; Pad to floppy disk.
times 1430 - ($-$$) db 0
; DOS sig and padding

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

dw 0xAA55

; Pad to floppy disk.
times 1430 - ($-$$) db 0