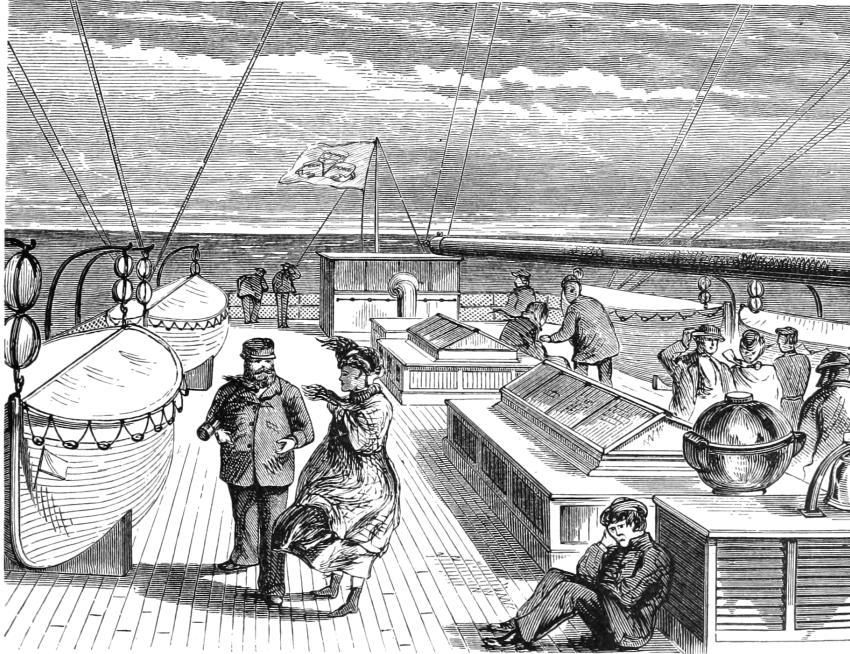# PoC||GTFO

IN A FIT OF STUBBORN OPTIMISM,
PASTOR MANUL LAPHROAIG
AND HIS CLEVER CREW
SET SAIL TOWARD
WELCOMING SHORES OF
THE GREAT UNKNOWN!

Heidelberg, Baden-Württemberg
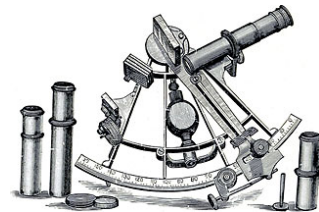
Это самиздат. Denn was man Schwarz auf Weiß besitzt, kann man getrost nach Hause tragen.
€0, $0 USD, £0, 0 RSD, 0 SEK, $50 CAD. `pocorgtfo11.pdf`. March 17, 2016.

Legal Note: Sony relies on the unsubstantiated residency of the unnamed defendant "Bushing" as a basis for California being the best forum. However, "Bushing" has not been identified, named, served, or connected with Mr. Hotz in any way that could warrant bringing the only identifiable defendant out to California. If "Bushing" does exist and can be ascertained at a later date, Sony would have to amend the complaint to properly name him/her which has not occurred. Thus, New Jersey is an alternative forum that exists to provide Sony with adequate relief. If Sony can obtain jurisdiction by merely including a hypothetical defendant by the name of "Bushing" that may live in California, then any Plaintiff can file suit in California and obtain jurisdiction by adding "Bushing" as a defendant.

Reprints: Bitrot will burn libraries with merciless indignity that even Pets Dot Com didn't deserve. Please mirror–don't merely link!–`pocorgtfo11.pdf` and our other issues far and wide, so our articles can help fight the coming robot apocalypse. We like the following mirrors.
`https://unpack.debug.su/pocorgtfo/`
`https://pocorgtfo.hacke.rs/`
`https://www.alchemistowl.org/pocorgtfo/`
`http://www.sultanik.com/pocorgtfo/`

Technical Note: Thanks to a Funky File Format Fire Sale, the file named `pocorgtfo11.pdf` is a polyglot in HTML, PDF, ZIP, and Ruby that executes as a quine over HTTP.

```
laphroaig% ruby pocorgtfo11.pdf
```

Printing Instructions: Pirate print runs of this journal are most welcome! PoC‖GTFO is to be printed duplex, then folded and stapled in the center. Print on A3 paper in Europe and Tabloid (11" x 17") paper in Samland. Secret government labs in Canada may use P3 (280 mm x 430 mm) if they like, but even the Americans on our staff will laugh at the use of awkward standards of measure. The outermost sheet should be on thicker paper to form a cover.

```
# This is how to convert an issue for duplex printing.
sudo apt-get install pdfjam
pdfbook --short-edge --vanilla --paper a3paper pocorgtfo11.pdf -o pocorgtfo11-book.pdf
```


|	
Preacherman	Manul Laphroaig
Editor of Last Resort	Melilot
TEXnician	Evan Sultanik
Editorial Whipping Boy	Jacob Torrey
Funky File Supervisor	Ange Albertini
Assistant Scenic Designer	Philippe Teuwen
and sundry others	

1 Please stand; now, please be seated.

Neighbors, please join me in reading this twelfth release of the International Journal of Proof of Concept or Get the Fuck Out, a friendly little collection of articles for ladies and gentlemen of distinguished ability and taste in the field of software exploitation and the worship of weird machines. This is our twelfth release, given on paper to the fine neighbors of Heidelberg.

If you are missing the first eleven issues, we the editors suggest pirating them from the usual locations, or on paper from a neighbor who picked up a copy of the first in Vegas, the second in São Paulo, the third in Hamburg, the fourth in Heidelberg, the fifth in Montréal, the sixth in Las Vegas, the seventh from his parents' inkjet printer during the Thanksgiving holiday, the eighth in Heidelberg, the ninth in Montréal, the tenth in Novi Sad or Stockholm, or the eleventh in Washington, D.C.

Our own Pastor Laphroaig opens this issue on page 4 by confessing to be a fan of junk hacking! He tells us to ignore the publicity and drama around a hack, to ignore even its target and its CVE. Instead, we should learn the mechanism of the hack, the clever tricks that make it work. Programming these mechanisms in nifty ways, be they ever so old, is surely not "junk"—think of it instead as an educational journey to far and exotic shores, on which this issue's great crew of authors stands ready to take you, neighbors!

In a fit of nostalgia for the good old vector arcade games, Trammel Hudson extended MAME to support native vector displays of the 1983 Star Wars arcade game on both his Tektronix 1720 scope and a Vectrex home vector display. Find it on page 6.
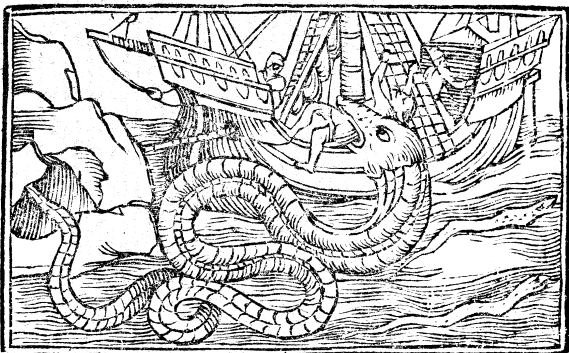
Eric Davisson contributes a 512-byte game for the PC BIOS on page 9. He discusses some nifty tricks for self-rewriting code in 16-bit Real Mode and shows that the fancier features of an operating system aren't needed to have a little fun—and that programming a constrained environment can be great fun indeed!

On page 15, Peter Ferrie describes his work toward a universal bypass for the E7 protection mode used on a number of Apple][disks. This is a follow up to his encyclopedic coverage of protection modes for this platform in PoC‖GTFO 10:7.

Ryan Speers and Travis Goodspeed have begun a series of tourist guides, intended to quickly introduce reverse engineers to a new platform. Page 20 provides a lightning-fast introduction to ARM's Cortex M series, which you'll find in modern devices with a megabyte or less of Flash memory. Page 28 contains similar notes for the Texas Instruments MSP430, MSP430X, and MSP430X2 architectures, a 16-bit competitor to the PIC and AVR.

At this journal, we generally frown upon defense, not because it is easy, but because it is so damned hard to describe properly. On page 24, Jeffrey Crowell presents a poor man's method of patching 32-bit x86 binaries to enforce the control flow graph. With examples in Radare2 and legible C, you'll be itching to write your own generic patchers for large binaries this weekend.

Page 33 describes how Evan Sultanik made this PDF—the one that you're reading—into a poyglot webserver quine in Ruby with its own самиздат PoC‖GTFO mirror.

It is with great sadness that we dedicate this release to the memory of our neighbor Ben Byer, the "hypothetical defendant by the name of 'Bushing'" who inspired many of us to put pwnage before politics, to keep on hacking. We're gonna miss him.

— — — —   — — —   — — — —

On page 40, the last page, we pass around the collection plate. We're not interested in your dimes, but we'd love some nifty proofs of concept. And remember, one hacker's "junk hacking" may hold the nifty tricks needed for another's treasured exploit!

2   In Praise of Junk Hacking

by Pastor Manul Laphroaig
in polite dissent to Daily Dave.
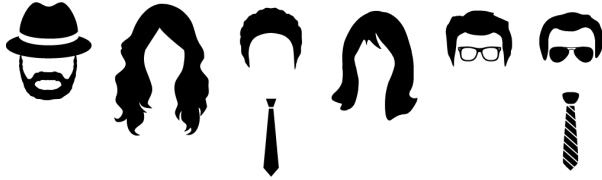
Gather round y'all, young and old, and listen to a story that I have to tell.

Back in 2014, when we were all eagerly waiting for `</SCORPION>` to debut on the TV network formerly known as the Columbia Broadcasting System, a minor ruckus was raised over Junk Hacking. The moral fiber of the youth, it was said, was being corrupted by a dozen cheap Black Hat talks on popping embedded systems with old bugs from the nineties. Who among us high-brow neighbors would sully the good name of our profession by hacking an ATM that runs Windows XP, when breaking into XP is old hat?

Let's think for just a minute and consider the best examples of neighborly junk hacking. Perhaps we'll find that rather than being mere publicity stunts, junk hacking is a way to step back from the daily grind of confidential consulting work, to share nifty tricks and techniques that are often more interesting than the bug itself.

— — — —   — — —   — — — —

Our first example today is from everyone's favorite doctor in a track suit, Charlie Miller. If you have the misfortune of reading about his work in the lay press, you might have heard that he could blow up laptop batteries by software,[1] or that he was recklessly irresponsible by disabling the power train of a car with a reporter inside.[2] That is to say, from the lay press articles, you wouldn't know a damned thing about what *mechanism* he experimented with.

So please, read the fucking paper, the battery hacking paper,[3] and ignore what CNN has to say on the subject. Read about how the Smart Battery Charger (SBC) is responsible for charging the battery even when the host is unresponsive, and consider how much more stable this would be than giving the host responsibility for managing the state. Read about how a complete development kit is available for the platform, about how the firmware update is flashed out of order to prevent bricking the battery.

Read about how the Texas Instruments BQ20Z80 chip is a CoolRISC 816 microcontroller, which was identified by Dion Blazakis through googling opcodes when the instruction set was not documented by the manufacturer. See that its mask ROM functions are well documented in `sluu225.pdf`.[4] Read about how code memory erases not to all ones, as most architectures would, but to `ff ff 3f` because that's a NOP instruction.

Read about how this architecture wasn't supported by IDA Pro, but that a plugin disassembler wasn't much trouble to write.[5] Read about how instructions on the CoolRISC platform are 22 bits wide and 24-bit aligned, so code might begin at any 3-byte boundary. See how Charlie bypasses the firmware checksums in order to inject his own code.

Can you really read all thirty-eight pages without learning one new trick, without learning anything nifty? Without anything more to say than your disappointment that batteries shipped with the default password? He who has eyes to read, let him read!

— — — —   — — —   — — — —

Loyal readers of this journal will remember PoC‖GTFO 2:4, in which Natalie Silvanovich gets remote code execution in a Tamagotchi's 6502 microcontroller through a plug-in memory chip. "Big whoop," some jerk might say, "local control of memory is getting root when you already have root!"
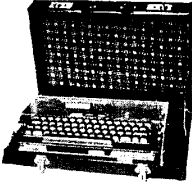
Re-read her article; it packs a hell of a lot into just two pages. The memory that she controls is just data memory, containing some fixed-size sprites and single byte describing the game that the cartridge should load. The game itself, like all other code, is already in the CPU's unwritable Mask ROM.

---

[1] If you RTFP, you'll note that the Apple batteries have a separate BQ29312 Analog Frontend (AFE) to protect against such nonsense, as well as a Matsushita MU092X in case the BQ29312 isn't sufficient.

[2] One time, my Studebaker ran out of gas on the highway. Maybe we should start a support group?

[3] `unzip pocorgtfo11.pdf batteryfirmware.pdf`

[4] `unzip pocorgtfo11.pdf sluu225.pdf`

[5] `unzip pocorgtfo11.pdf bq20z80.py`

So given just one byte of maneuverability, Natalie tried each value, discovering that a `switch()` statement had no `default` case, so values above `0x20` would cause a reboot, while really high values, above `0xD8`, would sometimes jump the game to a valid screen.

At this point she had a good idea that she was running off the end of a jump table, but as is common in the best junk hacking, she had no copy of the code and needed an exploit to extract the code. She did, however, know from die photographs and datasheets that the chip was a GeneralPlus GPLB52X with a 6502 instruction set. So she came up with the clever trick of making a *background picture* that, when loaded into LCD RAM, would form a NOP sled into shellcode that dumped memory out of an I/O port.

By reverse engineering that memory dump, she was able to replace her hail-Mary of a NOP sled with perfectly placed, efficient shellcode containing any number of fancy new features. You can even send your Tamagotchi to 30C3, if you like.

The point of her paper is no more about securing the Tamagotchi than Charlie's is about securing a battery. The point of the paper is to teach the reader the *mechanism* by which she dumped the firmware, and if you can read those two pages without learning something new about exploiting a target for which you have no machine code to disassemble, you aren't really trying. He who has eyes to read, let him read!
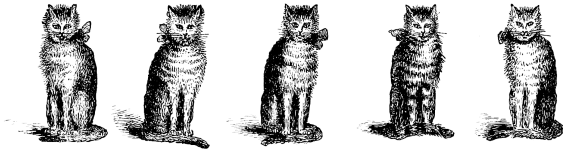
And this is the crux of the matter, dear neighbors. We become jaded by so much garbage on TV, so much crap in the news, and so many attempts to straight-jacket the narrative of security research by the mistaken belief that it must involve security. But the very best security research *doesn't* involve security! The very best research has no CVE, demands no patch, and has no direct relation to anything from your grandmother's credit card number to your server's `shadow` file.

The very best research is that which teaches you something new about the *mechanism* by which a machine functions. It teaches you how to build something, how to break something, or how to take something apart, but most of all it teaches you how the hell that thing really works.

So to hell with the target and to hell with the reporters. Teach me how a thing works, and teach me the techniques that you needed to do something clever with it. But if you casually dismiss the clever tricks learned from hacking an Apple][, a battery, or a Tamagotchi, I'm afraid that I'll have to ask you politely, but firmly, to get the fuck out.[6]

---

[6] Remember, though, that redemption is for everyone, and that one day you may find a strange and radiant machine you will treasure for the cleverness of its mechanisms, no matter if others call it junk. On that day we will welcome you back in the spirit of PoC!

3 Emulating Star Wars on a Vector Display

by Trammell Hudson
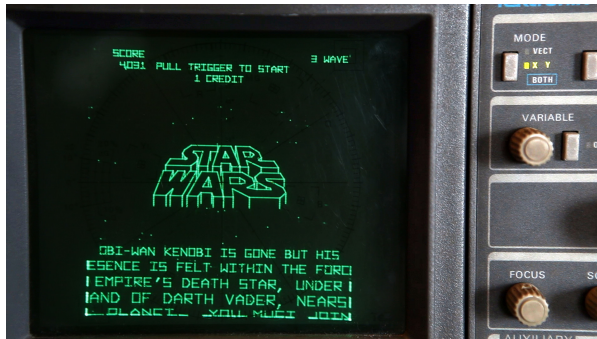
Star Wars was one of Atari's best vector games—possibly, the pinnacle of the golden age of arcade games. It featured 3D color vector graphics in an era w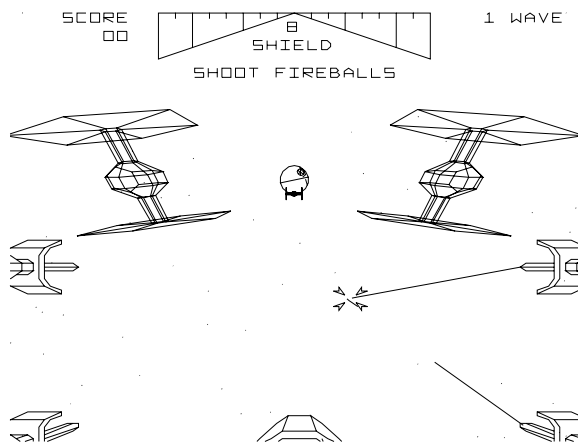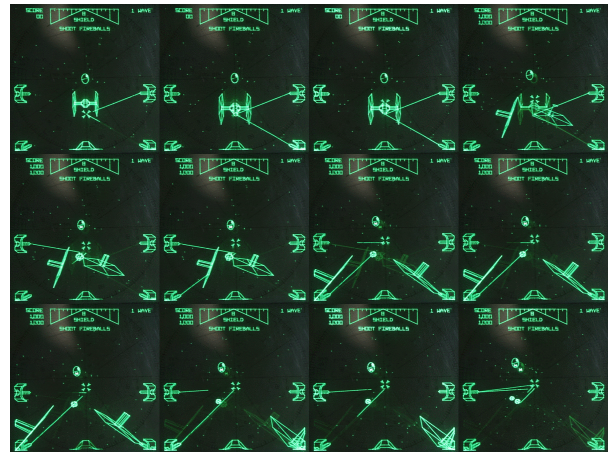hen most games were low-resolution bitmaps. It also had digitized voice samples from the movie, while its contemporary games were still using 8-bit beeps.

The graphics were 3D wireframe renderings that included features like the Tie fighters breaking up when they were hit by the player's lasers. There was no hidden wireframe removal; at this time it was not computationally feasible to do so.

3.1 Digital to Analog Converters

There were two common ways to generate the analog voltages to steer the electron beam in the vector monitor. Most early Atari games used the "Digital Voltage Generator," which used dual 10-bit DACs that directly output -2.5 to +2.5 volt signals. Starwars, however, used the "Analog Voltage Generator," in which the DACs generated the *slope* of the line, and opamps integrated the values to produce the output voltage. This is significantly more complex to emulate, and modern DACs and microcontrollers make it fairly easy to generate the analog voltages to drive the displays with resolution exceeding the precision of the old opamps.

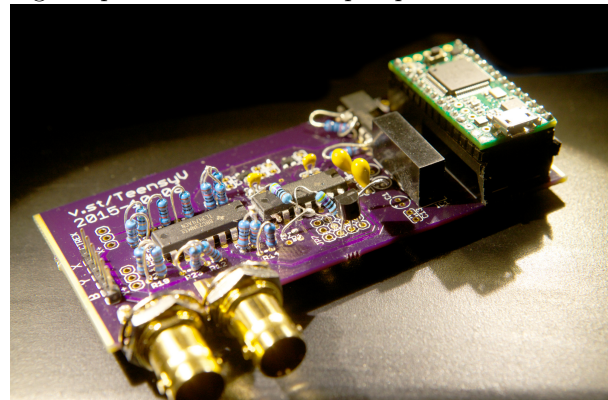The Starwars ROMs, along with almost all of Atari's vector games, can be emulated with MAME and the vectors extracted for display on actual vector hardware. Even though modern screens have exceeded the 10-bit resolution used by the game, the unique quality of a vector monitor is hard to convey. When compared to the low-resolution bitmap on a television monitor, the sharp lines and high resolution of the vectors are really stunning.

The open source hardware v.st quad-DAC boards output do 1.2 million samples per second, which is enough to steer the beam using Bresenham's line algorithm at a resolution of about 12 bits. While this is generating discrete points, the analog nature of the CRT means that smooth lines will be traced in the phosphor. The ARM's DMA engine clocks out the X and Y coordinates as well as the intensity, allowing the CPU to process incoming data from the USB serial connection without disrupting the output.
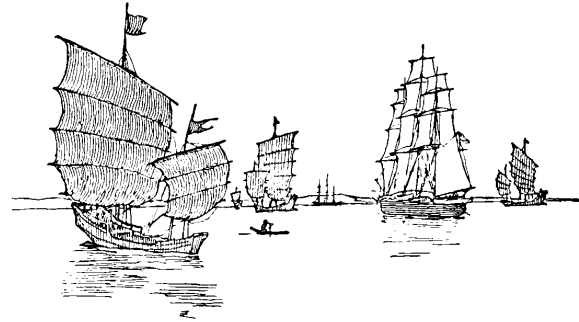
Source code for the v.st is available online or as an attachment to this PDF.[7]

3.2 Displays

Two inexpensive vector displays are the Tektronix 1720 vectorscope, a piece of analog NTSC video test equipment from a television studio, and the Vectrex, one of the only home vector console systems. The Tek uses an Electrostatic deflection CRT, which gives it very high bandwidth and almost instant transits between points, but at the cost of a very small deflection angle that results in a tiny screen and a very deep tube. The Vectrex has a magnetic deflection CRT, which allows it to be much shallower and significantly larger, but it requires many microseconds for the beam to stabilize in a new position. As a result, the DAC needs to take into account the "inertia" of the beam and wait for it to catch up.

3.3 Gameplay

Figure 2 compares the Tek 1720 on the left to the Vectrex on the right, which isn't very impressive on paper but will animate as a short video if you open `pocorgtfo11.pdf` in Adobe Reader. A longer video showing some of the different scenes is available. As the number of line segments increases, the slower display starts to flicker.

The game was played with a yoke, so the Y-axis mapping might seem backwards for a normal joystick. You can invert it in MAME by pressing Tab to bring up the config menu, selecting "Analog Controls" and "AD Stick Y Reverse".

While playing it on a small Vectrex or even smaller vectorscope doesn't quite capture the thrill of the arcade, it is quite fun to relive the vector art æsthetic at home and hear the digitized voice of Obi-Wan Kenobi telling you that "the Force will be with you, always."

---

[7] `git clone https://github.com/osresearch/vst`
`unzip pocorgtfo11.pdf vst.tar.bz2`
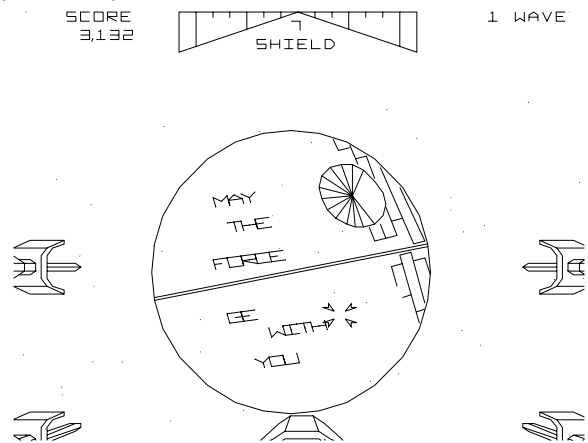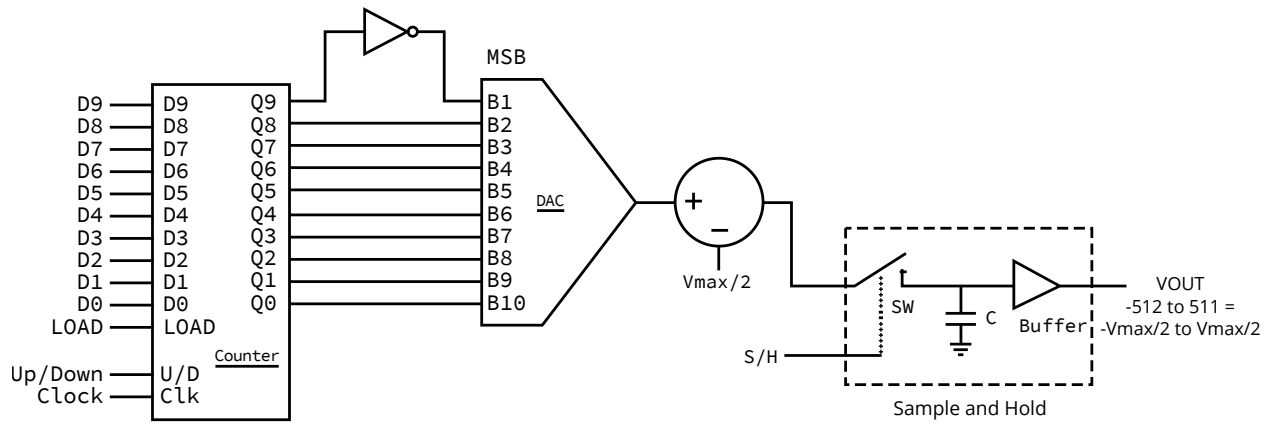
Figure 1 – Digital to Analog Signal Generator

Figure 2 – Tek 1720 vs Vectrex

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.
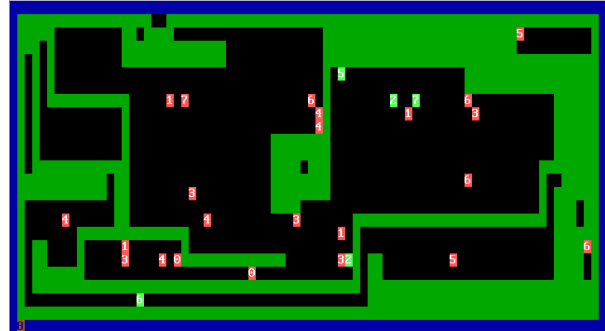
I found that there were only two small things I needed to do in order to write this code: make sure the boot image file is exactly 512 bytes and make sure the last two bytes are `0x55AA`. That's it! All the rest of the code is all yours. You could literally start a file with `0xEBFE` (two-byte unconditional 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 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 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.
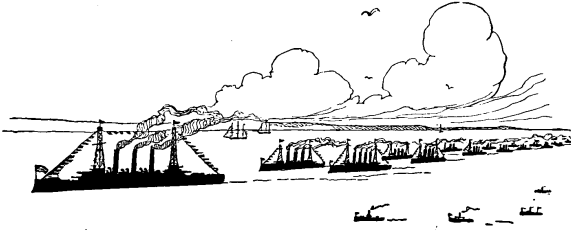
On a less technical note, this adds to the strategy of the game; it discourages just filling the screen up with the streak while avoiding items (so as to not create a mess) and just waiting out the clock. In fact, it is nearly impossible to win this way. To win, it is a better strategy to get as many 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!
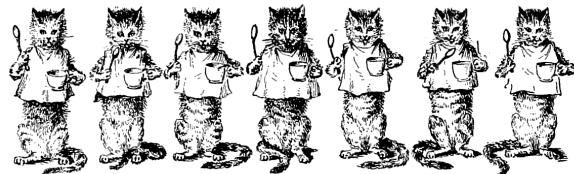
I hope these tricks are handy for you when writing your own 512-byte game, and also that you'll share your game with the rest of us. Complete code and prebuilt binaries are available in the ZIP portion of this release.[8]

---

[8]`unzip pocorgtfo11.pdf tronsolitare.zip`
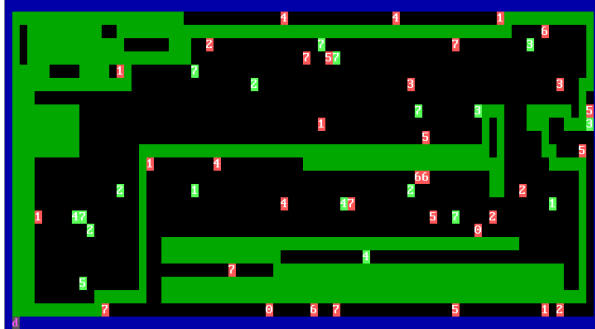
```
;Tron Solitare
;   *This is a PoC boot sector ( <512 bytes) game
;   *Controls to move are just up/down/left/right
;   *Avoid touching yourself, blue border, and the
;unlucky red 7

[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
```

12

```
            je penalty
185         pop ax              ;restore ax
            ret
187
        bonus:
189         mov byte [math], 0x01
                                ;make itemstuff: routine use
191                             ;add opcode
            call itemstuff
            stosw               ;put data back in
193         mov di, bx          ;restore coordinate
            add byte [selfmodify + 2], 3
195
            ret
197     penalty:
199         mov byte [math], 0x29
                                ;make itemstuff: routine use
201                             ;sub opcode
            call itemstuff
            cmp ax, 0xe000      ;sanity check for integer
203                             ;underflow
            ja underflow
205         stosw               ;put data back in
            mov di, bx          ;restore coordinate
207         ret
209
            underflow:
211            mov ax, 0x0100
               stosw
213            mov di, bx
               ret
215
        itemstuff:
217         pop dx              ;store return
            pop ax
219         and ax, 0x000f
            inc ax              ;1-8 instead of 0-7
221         shl ax, 8           ;multiply value by 256
            push ax             ;store the value
223
            mov bx, di          ;save coordinate
225         mov di, 0x0f02      ;set coordinate
            mov ax, [es:di]     ;read data at coordinate and
227                             ;subtract from score
            pop cx
229         math:
            add ax, cx          ;'add' is just a suggestion...
231         push dx             ;restore return
            ret
233
    score:
235     push di
        mov di, 0x0f02          ;set coordinate
237     mov ax, [es:di]         ;read data at coordinate
        ;for each mov of character, add 'n' to score
239         ;this source shows add ax, 1, however, each
            ;bonus item that is picked up increments this
241         ;value by 3 each time an item is picked up.
            ;Yes, this is self modifying code, which is
243         ;why the lable 'selfmodify:' is seen above, to
            ;be conveniently used as an address to pivot
245         ;off of in an add byte [selfmodify + offset to
            ;'1'], 3 instruction
247     selfmodify: add ax, 1 ;increment character in
                             ;coordinate
249     stosw               ;put data back in
        pop di
251     ;Why 0xf600 as score ceiling:
            ;if it was something like 0xffff, a score from
253         ;0xfffe would likley integer overflow to a low
            ;range (due to the progressive) scoring.
255         ;0xf600 gives a good amount of slack for this.
            ;However, it's still "technically" possible to
257         ;overflow; for example, hitting a '7' bonus
            ;item after already getting more than 171
259         ;bonus items (2048 points for bonus, 514
            ;points per move) would make the score go from
261         ;0xf5ff to 0x0001.
        cmp ax, 0xf600          ;is the score high enough to
263                             ;'win' ;#CHEAT
        ja win
265     ret
267 random:
        ;Decide whether to place bonus/trap
269        rdtsc
           and ax, 0x000f
271        cmp ax, 0x0007
           jne undo
273
        push cx                 ;save cx
275
        ;Getting random pixel
277     redo:
           rdtsc                ;random
279     xor ax, dx              ;xor it up a little
        xor dx, dx              ;clear dx
281     add ax, [0x046C]        ;moar randomness
        mov cx, 0x07d0          ;Amount of pixels on screen
283     div cx                  ;dx now has random val
        shl dx, 1               ;adjust for 'even' pixel values
```
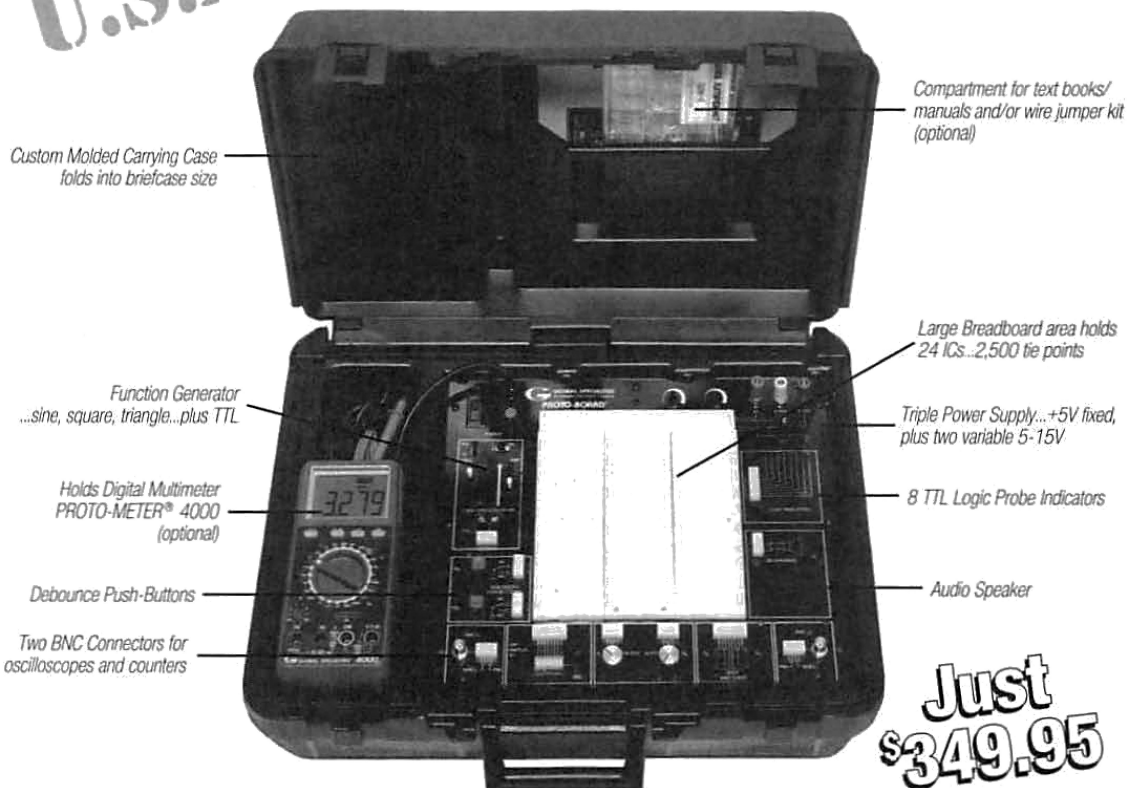
```
285     ;Are we clobbering other data?
        cmp dx, 0x0f02   ;Is the pixel the score?
287     je redo          ;Get a different value
289        push di          ;store coord
           mov di, dx
291        mov ax, [es:di]  ;read data at coordinate
           pop di           ;restore coord
293     cmp ax, 0x2f20   ;Are we on the snake?
        je redo
295     cmp ax, 0x1f20   ;Are we on the border?
        je redo
297
    ;Display random pixel
299     push di          ;save current coordinate
        mov di, dx       ;put rand coord in current
301
        ;Decide on item-type and value
303     powerup:
        rdtsc            ;random
305     and ax, 0x0007   ;get random 8 values
        mov cx, ax       ;cx has rand value
307     add cx, 0x5f30   ;baseline
        rdtsc            ;random
309     ;background either 'A' or 'C' (light green or
        ;red)
311        and ax, 0x2000   ;keep bit 13
           add ax, 0x5000   ;turn bit 14 and 12 on
313     add ax, cx       ;item-type + value
315     stosw            ;display it
        pop di           ;restore coordinate
317
    pop cx               ;restore cx
319
    undo:
321     ret
323 gameover:
        int 0x19         ;Reboot the system and restart
325                      ;the game.
327     ;Legacy gameover, doesn't reboot, just ends with
        ;red screen
329     ;xor di, di
        ;mov cx, 80*25
331     ;mov ax, 0x4f20
        ;rep stosw
333     ;jmp gameover
335 win:
        ;clear screen
337
        mov bx, [0x046C];Get timer state
339     add bx, 2
        delay2:
341        cmp [0x046C], bx
           jne delay2
343
        mov di, 0
345     mov cx, 0x07d0   ;enough for full screen
        winbg: mov ax, 0x0100
347                      ;xor ax, ax wont work, needs to
                         ;be this machine-code format
349     rep stosw        ;commit to video memory
351     mov di, 0x07c4   ;coord to start 'YOU WIN!' message
        xor cl, cl       ;clear counter register
353     winloop: mov al, [winmessage]
                         ;get win message pointer
355     mov ah, 0x0f     ;white text on black background
        stosw            ;commit char to video memory
357     inc byte [winloop + 0x01]
                         ;next character
359     cmp di, 0x07e0   ;is it the last character?
        jne winloop
361     inc word [winbg + 0x01]
                         ;incrememnt fill char/fg/bg
363                      ;(whichever is next)
        sub byte [winloop + 0x01], 14
365                      ;back to first character upon
                         ;next full loop
367     jmp win
369     winmessage:
        db 0x02, 0x20
371     dq 0x214e495720554f59    ;YOU WIN!
        db 0x21, 0x21, 0x20, 0x02
373
        ;BIOS sig and padding
375     times 510-($-$$) db 0
        dw 0xAA55
377
        ; Pad to floppy disk.
379     ;times (1440 * 1024) - ($ - $$)  db 0
```

13

HOME-WORK
For Electronics

MADE IN U.S.A.

Custom Molded Carrying Case folds into briefcase size

Compartment for text books/ manuals and/or wire jumper kit (optional)

Large Breadboard area holds 24 ICs...2,500 tie points

Function Generator ...sine, square, triangle...plus TTL

Triple Power Supply...+5V fixed, plus two variable 5-15V

Holds Digital Multimeter PROTO-METER® 4000 (optional)

8 TTL Logic Probe Indicators

Debounce Push-Buttons

Audio Speaker

Two BNC Connectors for oscilloscopes and counters

Just $349.95

Here's PB-503-C. It has every feature that our famous PB-503 offers, but we added one more, portability. Work on your projects at the office or school, take it home at night... it's for the engineer or student who wish to take their lab with them. **Instrumentation**, including a function generator with continuously variable sine, square, triangle wave forms and TTL pulses. **Breadboards** with 8 logic probe circuits. And a **Triple**

Power Supply with fixed 5VDC, plus two variable outputs (+5 to +15VDC). Throw-in 8 TTL compatible LED indicators, switches, pulsers, potentiometers, audio experimentation speaker... plus a life-time guarantee on all breadboarding sockets! And, because it's portable you will always have everything you need right in front of you! PB-503-C, one super test station for under $350! Order yours today!!

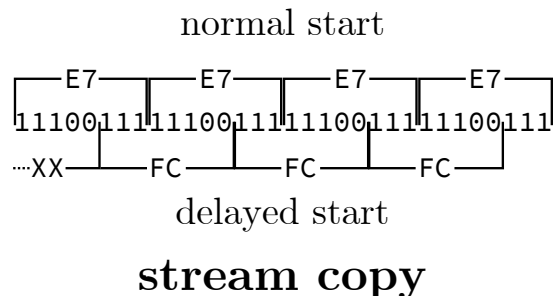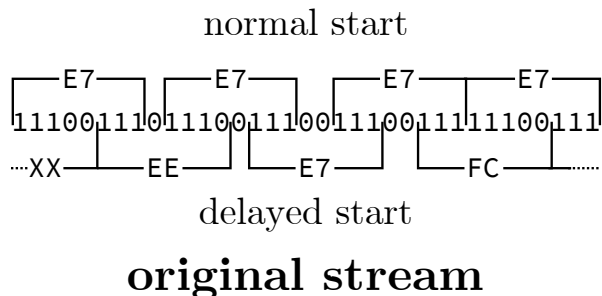**FOR MORE INFORMATION CALL 1-800-572-1028**

GLOBAL SPECIALTIES®
CIRCLE 182 ON FREE INFORMATION CARD

14

5 In Search of the Most Amazing Thing; or, Towards a Universal Method to Defeat E7 Protection on the Apple][Platform

by Peter Ferrie (qkumba, san inc)
with thanks to 4am

normal start

delayed start

original stream
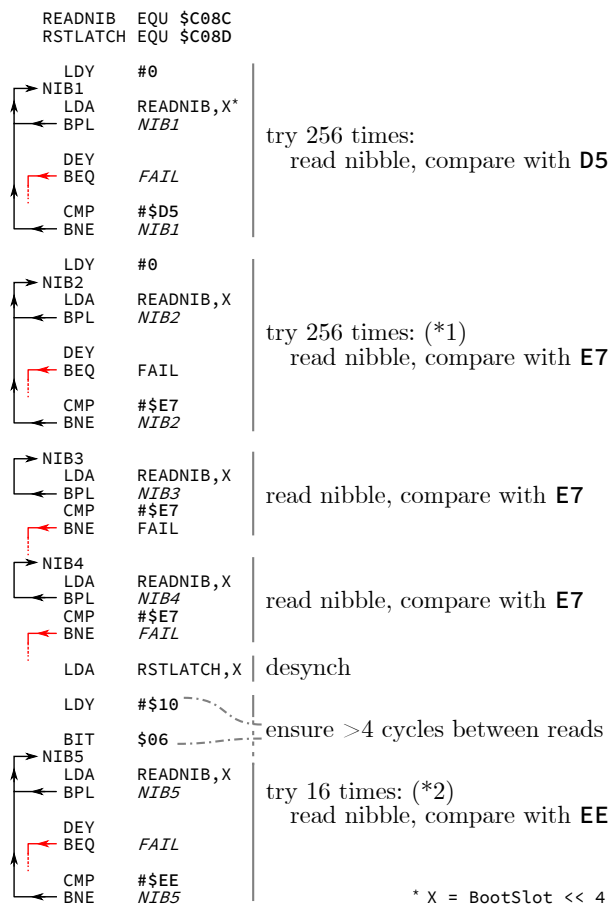
normal start

delayed start

stream copy

5.1 Introduction

In the early days, there was a protection technique known as the "generic bit-slip protection." In modern times, the cracker known as 4am has dubbed it the "E7 bitstream," because of the trigger values that are used to locate it. It was a very popular technique.

While many nibble-checks could be defeated simply by not allowing them to run at all, some protection routines required that the code be run to produce their side effects, such as to decrypt pages or to emit certain values that are checked later. At a high level, our goal is therefore to simulate the E7 bitstream entirely, allowing the protection routine to run as usual. That is, using a data-only solution to avoid making any changes to the code. Stated explicitly, our goal is to produce either disks that can be copied by COPYA (which, during a copy operation, converts nibble data to *sector data* and then back again) or ".dsk"-format disk images (which contain only sector data). Therefore, we need sector data that, when written to disk, produce *nibble data* that pass the protection check. For that to be possible, we must understand the protection itself and the code that uses it.

A primer on the hardware in general and this technique in particular was included in PoC∥GTFO 10:7. The theory is that after issuing an access of Q6H ($C08D+(slot*16)), the QA switch of the Data Register will receive a copy of the status bits, where it will remain accessible for four CPU cycles. After four CPU cycles, the QA switch of the Data Register will be zeroed. Meanwhile, assuming that the disk is spinning at the time, the Logic State Sequencer

continues to shift in the new bits. When the QA switch of the Data Register is zeroed, it discards the bits that were already shifted in, and the hardware will shift in bits as though nothing has been read previously. The relevant code looks like this:

```
READNIB   EQU $C08C
RSTLATCH  EQU $C08D

      LDY    #0
NIB1
      LDA    READNIB,X*       try 256 times:
      BPL    NIB1                 read nibble, compare with D5

      DEY
      BEQ    FAIL

      CMP    #$D5
      BNE    NIB1

      LDY    #0
NIB2
      LDA    READNIB,X
      BPL    NIB2             try 256 times: (*1)
                                 read nibble, compare with E7
      DEY
      BEQ    FAIL

      CMP    #$E7
      BNE    NIB2

NIB3
      LDA    READNIB,X
      BPL    NIB3             read nibble, compare with E7
      CMP    #$E7
      BNE    FAIL

NIB4
      LDA    READNIB,X
      BPL    NIB4             read nibble, compare with E7
      CMP    #$E7
      BNE    FAIL

      LDA    RSTLATCH,X       desynch

      LDY    #$10
                              ensure >4 cycles between reads
      BIT    $06
NIB5
      LDA    READNIB,X
      BPL    NIB5             try 16 times: (*2)
                                 read nibble, compare with EE
      DEY
      BEQ    FAIL

      CMP    #$EE
      BNE    NIB5                      * X = BootSlot << 4
```

Interestingly, the `bit $06` instruction is a misdirection. It exists only for the purpose of consuming some cycles. Any other instruction of equal duration could have been used, and it might be considered a watermark. While it is the value that exists most commonly, some titles changed the value of the address to `80` or `FF`, and these versions were spread, too.

In the most common implementation of the E7 protection, the stream on disk appears as `D5 E7 E7 E7 E7 E7 E7 E7 E7 E7 E7 E7 E7` with some harmless zero-bits in between. So from where do the other values come? The magic is in the timing of the reads, and timing is everything, so we must count the cycles!

```
LDA     READNIB,X
BPL     NIB4          2 cycles
CMP     #$E7          2 cycles
BNE     FAIL          2 cycles

LDA     RSTLATCH,X    4 cycles

LDY     #$10          2 cycles

BIT     $06           3 cycles
                     ──────────
                      15 cycles
```

One bit is shifted in every four CPU cycles, so a delay of 15 CPU cycles is enough for three bits to be shifted in. Those bits are discarded. However, since the CPU and the Disk][system are not synchronized, then depending on exactly when the initial read began, there can be up to two additional cycles in the total count. That puts us in the 16 cycle range, which is sufficient for a fourth bit to be shifted in and then discarded. In any case, the hardware sees it like this, due to a slip of three (or four) bits:

`D5 E7 E7 E7 [slip] EE E7 FC EE E7 FC EE EE FC`

In binary, the stream looks like this, with the seemingly redundant zero-bits in bold.

```
11010101    11100111    11100111    11100111
  D5           E7          E7          E7
11100111 0 11100111 00 11100111    11100111 0 11100111 00
  E7          E7          E7          E7         E7
11100111    11100111 0  11100111 0 11100111    11100111
  E7          E7          E7          E7          E7
```

However, by skipping the first three or four bits, the stream looks quite different.


```
  ⌐skipped
11100 11101110 0 11100111 00 11111100    11101110
       EE          E7          FC          EE
     0 11100111 00 11111100    11101110 0 11101110 0 11111100 111...
        E7          FC          EE          EE          FC
```

The old zero-bits are still in bold, and the newly exposed zero-bits are in italics. We can see that the old zero-bits form part of the new stream. This decodes to `EE E7 FC EE E7 FC EE EE FC`, and we have our magic values. The fourth bit must be a zero-bit in the original stream in case only three bits are slipped. Having the fifth bit be a zero-bit in the original stream makes a nice pattern of repeating values, if for no other reason.

5.2   Well-Groomed Data

In order to defeat this at all, we need to produce a regular 6-and-2 encoded sector which can be read by real hardware and copied by regular DOS.

We start by exploiting the point marked by (*1). There's a search for E7 after the D5. This allows us to introduce a full data prologue without breaking the check. So now we have this:

`D5 AA AD E7 E7 E7 E7 E7 E7 E7 E7 E7 E7 E7 E7 ...`

We can even conclude it with a regular epilogue so that there are no read errors. So now we have this:

`D5 AA AD E7 E7 E7 E7 E7 E7 E7 E7 E7 E7 E7 E7 ... DE AA`

It looks like a regular sector. The next step is to fill the stream with the appropriate values, including simulating the presence of the timing bits.

5.3   The Hard Stuff

We will use Bank Street Writer III for our first attempt, since it is the simplest example. Bank Street Writer III requires only one nibble from the pattern to be valid as an 8-bit decryption key for one page of memory. That nibble appears at a position four nibbles after the `EE`, and its value must be `E7`, so our pattern looks like this:

`EE ?? ?? ?? E7 ...`

Since we can't rely on timing bits in our stream (because we need *sector data* that produces *nibble*

data that this code interprets as valid), we can't place the `EE` inside a pair of `E7`s because after the bit-slip the wrong value will be read. Instead, we have to encode the value `EE` directly after discarding the first three bits, and placing a zero-bit in the fourth bit for compatibility purposes. In binary, that looks like this:

```
???01110 1110???? ???????? ????????
???????? 11100111 ...
```

After the bit-slip (and our extra zero-bit), the hardware sees:

```
...11101110 ???????? ???????? ????????
???? [11100111] ...
```

We must make those last four bits "disappear," in order to align our `E7` value correctly and allow it to be seen. If we turn those four bits into zeroes and distribute them within the stream, while adhering to the rule of not more than two consecutive zeroes, and replace the rest with ones, we get this:

```
...11101110 11111111 00 11111111 00
11111111 [11100111] ...
```

The hardware reads this as `EE FF FF FF E7`. Then we prepend one-bits and a zero-bit to the first (partial) nibble, like this:

```
[1110]11101110 11111111 00 11111111 00
11111111 [11100111] ...
```

After realigning the stream, we have this:

```
11101110 11101111 11110011 11111100
11111111 [11100111] ...
```

On disk, it appears as `EE EF F3 FC FF E7`.

The final step is to pad the data to a multiple of the sector size, so that we have a complete sector. We must also include the calculate the proper checksum. The remaining contents of the sector at this point are entirely arbitrary. We could place a text message or draw a picture, if we chose. Perhaps the most aesthetic version is to include a nibble which will zero the running value, and then fill the rest of the sector with `96`s, since `96` is the nibble value for zero. This will yield a sector which is devoid of all content other than the needed values. If that version is chosen, then a quick lookup in the nibble translation table shows us that the nibble value which will zero the running value is `F3`, so our whole stream appears as:

```
D5 AA AD E7 E7 E7 EE EF F3 FC FF E7 F3
96 96 ... DE AA
```

Great, it runs on hardware.

5.4 Apple for the Win, or Not.

Then we try AppleWin (as at 1.25.0.4). It doesn't work. Why not? Because instead of shifting bits into the data latch one at a time until the top bit is set, AppleWin shifts in an entire nibble immediately. It means that AppleWin does not (and cannot!) support bit-slip at all. Hmm, can we support both at the same time? Let's see about that.

We need to encode the first nibble as an `EE`, while also allowing a bit-slipping hardware to decode it as an `EE`. Well, we have that already, so we're halfway there! That just leaves the value four nibbles after the `EE`, which is currently the arbitrary value of `FF`. We change that `FF` to `E7`, so our stream on disk appears as:

```
EE EF F3 FC E7 E7
```

The final step is to pad the sector as we did previously. Using the aesthetic choice again, we zero the running value and then fill the rest of the sector with `96`s. A quick lookup in the nibble translation table shows us that the needed value is `D6`, so our whole stream appears as:

```
D5 AA AD E7 E7 E7 EE EF F3 FC E7 E7 D6
96 96 ... DE AA
```

We have a regular sector that works on hardware and AppleWin at the same time.

5.5 Totally Rad

Next up is Rad Warrior. It requires four nibbles from the pattern to be valid (as a 32-bit decryption key for four pages of memory), starting with the fourth nibble. It means that our Bank Street Writer III technique won't work because the pattern will be read differently between the bit-slip and the non-bitslip version, after the fourth nibble.

We have to come up with another technique. We do this by exploiting the point marked by (*2). There's a search for the `EE`. It means that we can insert nibbles after the point of the bit-slip, which will re-sync the stream to the non-slip form. At that point, we can insert any pattern that we need. We start with an arbitrary compatible sequence:

```
EF FF FF FF
```

In binary, it's:

`11101111 11111111 11111111 11111111`

After the bit-slip (and our extra zero-bit), the hardware sees:

`...11111111 11111111 11111111 1111`

As above, we must make those last four bits disappear, in order to align our pattern later. As above, we turn the four bits into zeroes and distribute them within the stream, while adhering to the rule of not more than two consecutive zeroes. Let's try this:

`...0 11111111 00 11111111 0 11111111`

The hardware reads this as `FF FF FF`. Then we prepend one-bits and a zero-bit to the first (partial) nibble again, like this:

`[1110]011111111 00 11111111 0 11111111`

After realigning the stream, we have this:

`11100111 11111001 11111110 11111111`

On disk, it appears as:

`E7 F9 FE FF`

That final `FF` is redundant, so we remove it. Then we append our complete pattern without any consideration for bit-slip. Our stream looks like this:

`E7 F9 FE EE E7 FC EE E7 FC EE EE FC`

The final step is to pad the sector as we did previously. Using the aesthetic choice again, we zero the running value and then fill the rest of the sector with `96`s. A quick lookup in the nibble translation table shows us that the needed value is `FB`, so our whole stream appears as:

`D5 AA AD E7 E7 E7 E7 F9 FE EE E7 FC EE E7 FC EE EE FC FB 96 96 ... DE AA`

We have a regular sector that works on hardware and AppleWin at the same time.

It also immediately supports Batman and Prince of Persia, both of which require the entire pattern (as a 64-bit decryption key for five pages of memory in Batman, and as a seed for several check-bytes during gameplay in Prince of Persia). Superb!

5.6   A Small Bump in the Road

Then we try it all in MAME (as of 0.169), because MAME is supposed to behave like the hardware. . . But. It. Does. Not. Work. Well, shit. And why not? Because while MAME does support bit-slip, it always consumes four bits for the code above, but most critically, it treats the bit in the fifth position as though it were always a one-bit.

It means that these four sequences are all decoded as `11111111 00 11111111 00` after the bit-slip. (Only one of which is correct.)

```
1 11111111  11110011  11111100
  11101111  11110011  11111100
3 11110111  11110011  11111100
  11100111  11110011  11111100
```

`11110011 11110011 11111100` is decoded as `10111111 00 11111111 00` after the bit-slip, which is not correct, either.

Despite the time that I've spent poring over the source code, I have not yet determined the cause, so we're left to work around it. Can we add support for MAME, while keeping the existing support? Without duplicating everything? Let's see about that.

We need to move a zero-bit beyond the slipped region so that the hardware will read the same bits that MAME does.

```
  [1110]0  11111111  00  11111111  0x  ...
2       V——>——>——>——>——>——>——^
```
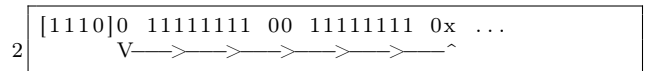
After moving the zero bit, we have `[1110]11111111 00 11111111 00 ....` Realigning that stream, we get `11101111 11110011 11111100 ...`, which looks good. On disk, it appears as `EF F3 FC`.

Then we append our complete pattern without any consideration for bit-slip. This stream is `EF F3 FC EE E7 FC EE E7 FC EE EE FC`.

The final step is to pad the sector as we did previously. Using the aesthetic choice again, we zero the running value and then fill the rest of the sector with `96`s. A quick lookup in the nibble translation table shows us that the needed value is `EA`, so our whole stream appears as `D5 AA AD E7 E7 E7 EF F3 FC EE E7 FC EE E7 FC EE EE FC EA 96 96 ... DE AA`.

18

5.7 Success!

We have a truly universal nib sequence, which works on hardware, which works on AppleWin, which works on MAME (and which will still work when the bug is fixed), and which defeats the `E7` protection.

Here is our universal sequence in the form of a disk sector:

```
    03 00 03 02 02 02 00 03 03 01 02 02 00 02 02 00
 2  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 4  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 6  00 00 00 00 00 00 01 00 01 01 03 00 00 01 02 02
    03 00 00 00 03 00 00 00 00 00 00 00 00 00 00 00
 8  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
10  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    00 00 00 00 00 00 00 00 00 00 00 00 01 00 01 02
12  01 02 01 00 03 00 01 02 01 02 01 00 00 00 00 00
    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
14  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
16  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

This can be applied wherever the `E7` sequence is the regular pattern. For other patterns, such as those used by Thunder Mountain's "Dig Dug" (`E7 EE EE EE E7 E7 EE E7 EE EE EE E7 EE E7 EE EE`), Sunburst's "1-2-3 Sequence Me" (`BB F9 Fx`), and MCE's "The 4th R - Reasoning" (`EB B6 EF 9A DB B7 ED F9 D7 BF BD A7 B3 FF B3 BA`), just place the proper pattern after the "EF F3 FC" sequence, pad the sector as you like, and then fix the sector checksum.

For the record, the E7 stream is used in many other titles (games or educational software), such as Commando, Deathsword, Ikari Warriors, Impossible Mission II, Karate Champ, Paperboy, Rambo First Blood Part II (a pure text adventure!), Summer/Winter/World Games, The Ancient Art of War [at Sea], Tetris, and Xevious.

As far as we know, this technique first appeared in 1983. It was used to protect the title Locksmith, ironically a product for defeating copy-protection.

None of the disk copiers of the day could copy E7 disks without a parameter unique to the target, so duplicating these disks always required a bit of expertise.

5.8 Final Words

Here is an interesting question: What if you don't have an entire sector available on the track that you need?

Fortunately, this would be a concern only for a protection which used the rest of the sector (and the rest of the track) for meaningful data, which I have not seen so far. In any case, the solution would be to insert only the nibble sequence "`EF F3 FC ... EE EE FC`" and to not pad the sector. This would yield a freely-copyable disk in its original form. However, we must discourage that idea with these words from 4am:

Never patch an original disk.
Don't reduce the number of original disks in the world.
They aren't making any more of them.

-4am

6 A Tourist's Phrasebook for Reversing Embedded ARM in the Dialect of the Cortex M Series

by Travis Goodspeed and Ryan Speers

Ahoy there, neighbor!

Welcome to another installment of our series of quick-start guides for reverse engineering embedded systems. Our goal here is to get you situated with the architecture of smaller devices as quickly as possible, with a minimum of fuss and formality.

Those of you who have already worked with ARM might find it to be a useful refresher, while those of you new to the architecture will find that it isn't really as strange as you've been led to believe. If you've already reverse engineered binaries for any platform, even x86 Windows applications, you'll soon feel right at home.

We've written this guide with STM32 devices for specific examples, but with minor differences it applies well enough to the Cortex M series as a whole. These devices generally have a megabyte or less of Flash and at most a few hundred kilobytes of RAM. By and large, they only run the Thumb2 instruction set, without support for the older AARCH32 instruction set. For larger ARM chips, such as those used in smartphones and tablets, you might be better served by a different introduction.

6.1 At a Glance

Common Models
STM32, EFM32

Architecture
32-bit registers
16-bit and 32-bit Thumb(2) instructions

Registers
R15: Program Counter
R14: Link Register
R13: Stack Pointer
R0 to R12: General Use

6.2 Basics of the Instruction Set

Back in the day, ARM used fixed-width 32-bit RISC instructions. Like the creation of the world, this was widely regarded as a mistake, and many angry people wrote comments complaining that it was a waste of space, and that RISC wouldn't "change everything." These instructions were always 32-bit word aligned, so the lowest two bits of the Program Counter (R15) were always zero.

Larger ARM chips, such as those in an early smartphone, support two instructions sets. If the least significant bit of the program counter is clear (0), then the 32-bit instruction set is used, whereas if that bit is set (1), the chip will use a 16-bit instruction set called Thumb. Registers are still 32 bits wide, but the instructions themselves are only a half-word. They must be half-word aligned.

Because Thumb instructions have fewer bits to spare, code in larger ARM machines will switch between ARM and Thumb as it is convenient. You can see this in the least significant bit of a function pointer, where an ARM function's address will be even, while a Thumb function's address will be odd.

The Cortex M3 devices speak a slimmer dialect than the big-iron ARM chips. This dialect drops the 32-bit wide instruction set entirely, supporting only Thumb and Thumb2 instructions.[9] Because of this, all functions and all interrupt handlers are referred to by *odd* addresses, which are actually the address of the byte *after* the real starting address! If you see a call to `0x08005615`, that is really a call to the Thumb code at `0x08005614`.

6.3 Registers and Calling Convention

Arguments are passed to the child function from R0 to R3. R4 to R11 hold local variables, and the child function *must* restore them before returning to the parent function. Values are returned in R0 to R3, and these registers are not preserved by the child.

Much like in PowerPC and very unlike x86, the Link Register (R14, a.k.a. LR) holds the return address. A leaf function, having no children, might never write its return pointer to the stack. The `BL` instruction automatically moves the old Program Counter into the Link Register when calling a child, so parent functions must manually save R14 before calling children. The return instruction, `BLR`, functions by moving R14 (LR) into R15 (PC).

---

[9]Thumb2 instructions run from Thumb mode. The only thing new about them is that they can be longer than 16 bits, so your disassembler might be slightly confused about their starting position.
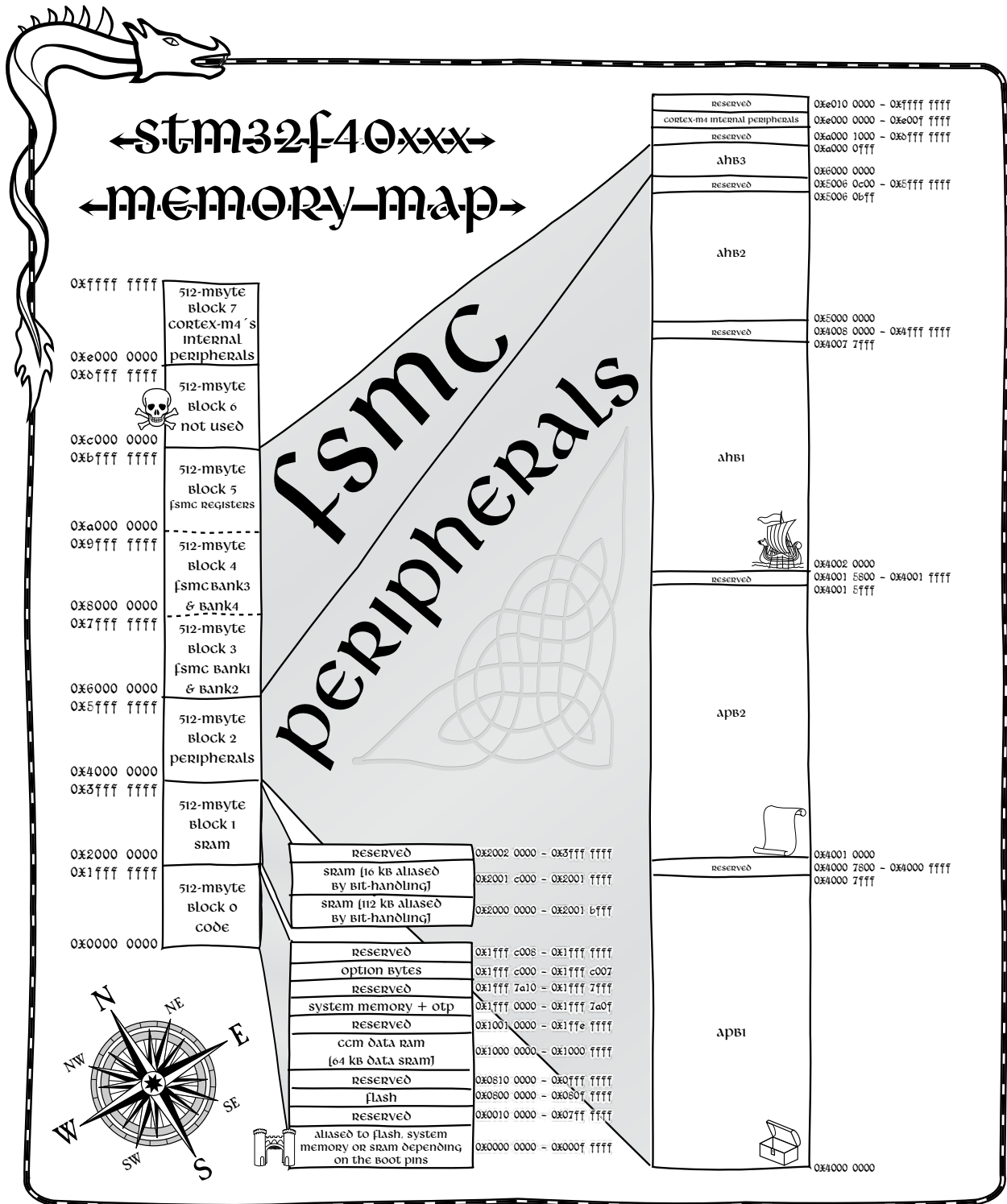
stm32f40xxx memory map

fsmc peripherals

Address	Block
0xffff ffff	512-mbyte block 7 cortex-m4´s internal peripherals
0xe000 0000	
0xdfff ffff	512-mbyte block 6 not used
0xc000 0000	
0xbfff ffff	512-mbyte block 5 fsmc registers
0xa000 0000	
0x9fff ffff	512-mbyte block 4 fsmc bank3 & bank4
0x8000 0000	
0x7fff ffff	512-mbyte block 3 fsmc bank1 & bank2
0x6000 0000	
0x5fff ffff	512-mbyte block 2 peripherals
0x4000 0000	
0x3fff ffff	512-mbyte block 1 sram
0x2000 0000	
0x1fff ffff	512-mbyte block 0 code
0x0000 0000	

Region	Range
reserved	0x2002 0000 – 0x3fff ffff
sram (16 kb aliased by bit-handling)	0x2001 c000 – 0x2001 ffff
sram (112 kb aliased by bit-handling)	0x2000 0000 – 0x2001 bfff

Region	Range
reserved	0x1fff c008 – 0x1fff ffff
option bytes	0x1fff c000 – 0x1fff c007
reserved	0x1fff 7a10 – 0x1fff 7fff
system memory + otp	0x1fff 0000 – 0x1fff 7a0f
reserved	0x1001 0000 – 0x1ffe ffff
ccm data ram (64 kb data sram)	0x1000 0000 – 0x1000 ffff
reserved	0x0810 0000 – 0x0fff ffff
flash	0x0800 0000 – 0x080f ffff
reserved	0x0010 0000 – 0x07ff ffff
aliased to flash, system memory or sram depending on the boot pins	0x0000 0000 – 0x000f ffff

Region	Range
reserved	0xe010 0000 – 0xffff ffff
cortex-m4 internal peripherals	0xe000 0000 – 0xe00f ffff
reserved	0xa000 1000 – 0xdfff ffff
ahb3	0xa000 0fff … 0x6000 0000
reserved	0x5006 0c00 – 0x5fff ffff / 0x5006 0bff
ahb2	0x5000 0000
reserved	0x4008 0000 – 0x4fff ffff / 0x4007 7fff
ahb1	0x4002 0000
reserved	0x4001 5800 – 0x4001 ffff / 0x4001 5fff
apb2	0x4001 0000
reserved	0x4000 7800 – 0x4000 ffff / 0x4000 7fff
apb1	0x4000 0000

N NE E SE S SW W NW

Figure 3 – STM32F40xxx Memory Map

6.4 Memory Map

Figure 3 shows the memory layout of the STM32F405, a Cortex M4 device. Study this map for a moment, before we go on to how to use it in your adventure!

Because Cortex M devices have four gigabytes of address space but hardly a megabyte of Flash, they keep functionally different parts of memory at very different addresses.

Code memory is officially the range from `0x00000000` to `0x1FFFFFFF`, but in nearly all cases, you'll find that Flash is mapped to begin at `0x0800-0000`. When reverse engineering an application, you'll find that it's either written here or a few dozens of kilobytes later, to leave room for a bootloader.

SRAM is usually mapped to begin at `0x2000-0000`, so it's safe to assume that any read or write to an absolute address in this region is a global variable, and also that the stack and heap fit somewhere in this range. Unlike a desktop application, which loads its initial globals directly into a `.data` segment, an embedded application must manually initialize its data variables, possibly by copying a large chunk from Flash into SRAM.

Peripheral memory begins at `0x40000000`. Both because peripherals are most often referred to by an explicit address, and because Flash comes with no linking systems or system calls, reads and writes to this region are a gold mine for a reverse engineer!

System control registers are at `0xE0000000`. These are used to do things like moving the interrupt table or reading the chip's model number.

6.5 Making Sense of Pointers

Let us teach you some nifty tricks about pointers in Thumb machines.

Back when ARM was first designed, 32-bit fixed-width instructions with 32-bit alignment were all the rage, and all the cool kids (POWER, SPARC, Alpha) used them. Later on, when the Thumb instruction set was being designed, its designers chose 16-bit instructions that could be mapped back to the same 32-bit core. The CPU would fetch a 32-bit ARM instruction if the least-significant bit of the program counter were even, and a 16-bit Thumb instruction if the program counter were odd.

But these Cortex chips generally ship just Thumb and Thumb2, without backward compatibility to 32-bit ARM instructions. So the trick, which

you can try in the next section, is that data pointers are always even and instruction (function) pointers are always odd.

6.6 Making Sense of the Interrupt Table

Let's take a look at the interrupt table from the beginning of a Cortex M firmware image. These are 32-bit little endian addresses, which are to be read backwards.

```
0000000  30 14 00 20    21 41 00 08
2        39 57 00 08    3d 57 00 08
0000010  41 57 00 08    45 57 00 08
4        49 57 00 08    00 00 00 00
0000020  00 00 00 00    00 00 00 00
6        00 00 00 00    51 57 00 08
0000030  4d 57 00 08    00 00 00 00
8        55 57 00 08    59 57 00 08
0000040  ...
```

Note that the first word, `0x20001430`, is in the SRAM region; this is because the first word of a Cortex M interrupt table is the initialization value for the Stack Pointer (R13). The second word, `0x0800-4121`, is the initialization value for the Program Counter (R15), so we know the entry point of the application is Thumb2 code starting at `0x08004120`.

Except for some reserved (zeroed) words, the handler addresses are all in Flash memory and represent the interrupt handler functions. We can look up the meaning of each handler in the specific chip's programming guide, then chase the ones that are most relevant. For example, if we are reverse engineering a USB device, powered by an STM32F3xx, the STM32F37xx reference manual tells us that the interrupts at offsets `0x000000D8` and `0x0000001C` handle USB events. These might be good handlers to reverse early in the process.

6.7 Loading into IDA Pro or Radare2

To load the application into IDA Pro or Radare2, you generally need to know the loading point and the locations of some other memories.

The loading point will be at or near `0x08000000`, depending upon whether a bootloader comes before your image. If you are working from a JTAG dump, just use the address the image came from. If you are working from a `.dfu` (Device Firmware Update) file, it will contain a loading address in its header metadata.

When given a raw dump without a starting address, disassemble the instructions and try to find a loading address at which the interrupt handlers line up. (The interrupt vector table is usually at `0x08000000` at boot, but it can be moved to a new address by software.)

6.8 Making Sense of the Peripherals

The Cortex M3 contains two peripheral regions. At `0x40000000`, you will find the most useful ones for reverse engineering applications, such as UART and USB controllers, General Purpose IO (GPIO), and other devices. Unfortunately, these peripherals are not generic to the Cortex M3 as an architecture; rather, they are specific to each individual chip.

Supposing you are reverse engineering an application for the STM32F3xx series, you would download the Peripheral Support Library for that chip from its manufacturer and eventually find yourself reading `stm32f30x.h`. For other chips, there are similar headers, each of which is written around C structs for register groups and preprocessor definitions for peripheral base addresses and offsets.

Suppose we know from reverse engineering a circuit board that USART2 is used by our target application to send packets to a radio chip, and we would like to search for all functions that use this peripheral. Working backwards, we find the following relevant lines in `stm32f30x.h`.

```
1  //Abbreviated USART register struct.
   typedef struct{
3      __IO uint32_t CR1;      //+0x00
       __IO uint32_t CR2;
5      __IO uint32_t CR3;
       __IO uint16_t BRR;
7    uint16_t  RESERVED1;
       __IO uint16_t GTPR;
9    uint16_t  RESERVED2;
       __IO uint32_t RTOR;
11     __IO uint16_t RQR;
     uint16_t  RESERVED3;
13     __IO uint32_t ISR;
       __IO uint32_t ICR;
15     __IO uint16_t RDR;      //+0x24 RX Data Reg
     uint16_t  RESERVED4;
17     __IO uint16_t TDR;      //+0x28 TX Data Reg
     uint16_t  RESERVED5;
19 } USART_TypeDef;

21 //USART location definitions.
   #define USART2                              \
23         ((USART_TypeDef *) USART2_BASE)
```

```
   #define USART2_BASE                         \
25         (APB1PERIPH_BASE + 0x00004400)
   #define APB1PERIPH_BASE                     \
27         PERIPH_BASE
   #define PERIPH_BASE                         \
29         ((uint32_t)0x40000000)
```

This means that USART2's data structure is located at `0x40004400`. From the `USART_TypeDef` structure, we know that data is received from USART2 by reading `0x40004424` and written to USART2 by writing to `0x40004428`! Searching for these addresses ought to easily find us the read and write functions for that port.

6.9 Other Oddities

Please note that this guide has left out some features unique to the STM32 series, and that each chip has its own little quirks. You'll find different memory maps on each implementation, and anything that looks confusing is likely worth spending more time to understand.

For example, some ARM devices offer Core-Coupled Memory (CCM), which is SRAM that's wired directly to the CPU's internal data bus rather than to the main memory bus of the chip. This makes fetches lightning fast, but has the complications that the memory is unusable for DMA or code fetches. Care for a non-executable stack, anyone?

Another quirk is that many devices map the same physical memory to multiple virtual locations. In some high-performance code, the use of both cached and uncached memory can allow for more efficient operation.

Additionally, address zero often contains a duplicate of the boot memory, which is usually Flash but might be executable SRAM. Presumably this was done to allow for code that has compatible immediate addresses when booting from either memory, but PoC‖GTFO 10:8 describes a nifty little jailbreak that relies on dumping the 48K recovery bootloader of an STM32F405 chip out of Flash through a null-pointer read.

——— ——— ———

We hope that you've enjoyed this friendly little guide to the Cortex M3, and that you'll keep it handy when reverse engineering firmware from that platform.

7 A Ghetto Implementation of CFI on x86

by Jeffrey Crowell

In 2005, M. Abadi and his gang presented a nifty trick to prevent control flow hijacking, called *Control Flow Integrity*. CFI is, essentially, a security policy that forces the software to follow a predetermined control flow graph (CFG), drastically restricting the available gadgets for return-oriented programming and other nifty exploit tricks.

Unfortunately, the current implementations in both Microsoft's Visual C++ and LLVM's clang compilers require source to be compiled with special flags to add CFG checking. This is sufficient when new software is created with the option of added security flags, but we do not always have such luxury. When dealing with third party binaries, or legacy applications that do not compile with modern compilers, it is not possible to insert these compile-time protections.

Luckily, we can combine static analysis with binary patching to add an equivalent level of protection to our binaries. In this article, I explain the theory of CFI, with specific examples for patching x86 32-bit ELF binaries—without the source code.

CFI is a way of enforcing that the intended control flow graph is not broken, that code always takes intended paths. In its simplest applications, we check that functions are always called by their intended parents. It sounds simple in theory, but in application it can get gnarly. For example, consider:

```
1 int a() { return 0; }
  int b() { return a(); }
3 int c() { return a() + b() + 1; }
```

For the above code, our pseudo-CFI might look like the following, where `called_by_x` checks the return address.

```
1 int a() {
    if (!called_by_b && !called_by_c) {
3     exit();
    }
5   return 0;
  }
7 int b() {
    if (!called_by_c) {
9     exit();
    }
11  return a();
  }
13 int c() { return a() + b() + 1; }
```

Of course, this sounds quite easy, so let's dig in a bit further. Here is a very simple example program to illustrate ROP, which we will be able to effectively kill with our ghetto trick.

```
1 #include <string.h>

3 void smashme(char* blah) {
    char smash [16];
5   strcpy(smash, blah);
  }
7
  int main(int argc, char** argv) {
9   if (argc > 1) {
      smashme(argv[1]);
11  }
  }
```

In x86, the stack has a layout like the following.

Local Variables
Saved ebp
Return Pointer
Parameters
. . .

By providing enough characters to `smashme`, we can overwrite the return pointer. Assume for now, that we know where we are allowed to return to. We can then provide a whitelist and know where it is safe to return to in keeping the control flow graph of the program valid.

Figure 4 shows the disassembly of `smashme()` and `main()`, having been compiled by GCC.

Great. Using our whitelist, we know that `smashme` should only return to 0x08048456, because it is the next instruction after the `ret`. In x86, `ret` is equivalent to something like the following. (This is not safe for multi-threaded operations but we can ignore that for now.)

```
1 pop ecx; puts the return address to ecx
  jmp ecx; jumps to the return address
```

```
[0x08048320]> pdf@sym.smashme
/ (fcn) sym.smashme 26
|              ; arg int arg_2        @ ebp+0x8
|              ; var int local_6      @ ebp-0x18
|              ; CALL XREF from 0x08048451 (sym.smashme)
|           0x0804841d      55              push ebp
|           0x0804841e      89e5            mov ebp, esp
|           0x08048420      83ec28          sub esp, 0x28
|           0x08048423      8b4508          mov eax, dword [ebp+arg_2]   ; [0x8:4]=0
|           0x08048426      89442404        mov dword [esp + 4], eax
|           0x0804842a      8d45e8          lea eax, [ebp-local_6]
|           0x0804842d      890424          mov dword [esp], eax
|           0x08048430      e8bbfeffff      call sym.imp.strcpy
|           0x08048435      c9              leave
\           0x08048436      c3              ret
[0x08048320]> pdf@sym.main
/ (fcn) sym.main 33
|              ; arg int arg_0_1       @ ebp+0x1
|              ; arg int arg_3         @ ebp+0xc
|              ; DATA XREF from 0x08048337 (sym.main)
|              ;-- main:
|           0x08048437      55              push ebp
|           0x08048438      89e5            mov ebp, esp
|           0x0804843a      83e4f0          and esp, 0xfffffff0
|           0x0804843d      83ec10          sub esp, 0x10
|           0x08048440      837d0801        cmp dword [ebp + 8], 1        ; [0x1:4]=0x1464c45
|      ,=<  0x08048444      7e10            jle 0x8048456
|      |    0x08048446      8b450c          mov eax, dword [ebp+arg_3]   ; [0xc:4]=0
|      |    0x08048449      83c004          add eax, 4
|      |    0x0804844c      8b00            mov eax, dword [eax]
|      |    0x0804844e      890424          mov dword [esp], eax
|      |    0x08048451      e8c7ffffff      call sym.smashme
|      |       ; JMP XREF from 0x08048444 (sym.main)
|      `->  0x08048456      c9              leave
\           0x08048457      c3              ret
```

Figure 4 – Disassembly of `main()` and `smashme()`.

Cool. We can just add a check here. Perhaps something like this?

```
pop ecx; puts the return address to ecx
2 cmp ecx, 0x08048456; check that we return to
       the right place
  jne 0x41414141; crash
4 jmp ecx; effectively return
```

Now just replace our `ret` instruction with the check. `ret` in x86 is simply this:

```
$ rasm2 -a x86 -b32 "ret"
2 c3
```

where our code is this:

```
$ rasm2 -a x86 -b32 "pop ecx;cmp ecx, 0
    x08048456; jne 0x41414141; jmp ecx"
2 5981f9568404080f8534414141ffe1
```

Sadly, this will not work for several reasons. The most glaring problem is that `ret` is only one byte, whereas our fancy checker is 15 bytes. For more complicated programs, our checker could be even larger! Thus, we cannot simply replace the `ret` with our code, as it will overwrite some code after it—in fact, it would overwrite `main`. We'll need to do some digging and replace our lengthy code with some relocated parasite, symbiont, code cave, hook, or detour—or whatever you like to call it!

Nowadays there aren't many places to put our code. Before x86 got its no-execute (NX) MMU bit, it'd be easy to just write our code into a section like `.data`, but marking this as `+x` is now a huge security hole, as it will then be `rwx`, giving attackers a great place for putting shellcode. The `.text` section, where the main code usually goes, is marked `r-x`, but there's rarely slack space enough in this section for our code.

Luckily, it's possible to add or resize ELF sections, and there're various tools to do it, such as *Elfsh*, *ERESI*, etc. The challenge is rewriting the appropriate pointers to other sections; a dedicated tool for this will be released soon. Now we can add a new section that is marked as `r-x`, replace our `ret` with a jump to our new section—and we're ready to take off!

Well, wheels aren't up yet. As mentioned before, `ret` is `c3`, but absolute jumps are five bytes.

```
$ rasm2 -a x86 -b32 "jmp 0x41414141"
2 e93c414141
```

So what is left to do? Well, we can simply rewind to the first complete opcode five bytes before the `ret`, and add a jump, then relocate the remaining opcodes. In this case, we could do something like this:

```
  smashme:
2 push ebp
  mov ebp, esp
4 sub esp, 0x28
  mov eax, dword [ebp + 8]
6 mov dword [esp + 4], eax
  lea eax, [ebp - 0x18]
8 mov dword [esp], eax
  jmp parasite
10
  parasite:
12 call sym.imp.strcpy
  leave
14 pop ecx
  cmp ecx, 0x08048456
16 jne 0x41414141
  jmp ecx
```

Here, `parasite` is mapped someplace else in memory, such as our new section.

With this technique, we'll still to have to pass on protecting a few kinds of function epilogues, such as where a target of a jump is within the last five bytes. Nevertheless, we've covered quite a lot of the intended CFG.

This approach works great on platforms like ARM and MIPS, where all instructions are constant-length. If we're willing to install a signal handler, we can do better on x86 and amd64, but we're approaching a dangerous situation dealing with signals in a generic patching method, so I'll leave you here for now. The code for applying the explained patches is all open source and will soon be extended to use emulation to compute relative calls.

Thanks for reading!
Jeff

27

8 A Tourist's Phrasebook for Reversing MSP430

by Ryan Speers and Travis Goodspeed

Howdy, y'all!

Welcome to another installment of our series of quick-start guides for reverse engineering embedded systems. Our goal here is to get you situated with the MSP430 architecture as quickly as possible, with a minimum of fuss and formality.

Those of you who have already used an MSP430 might find this to be a useful reference, while those of you new to the architecture will find that it isn't really all that strange. If you've already reverse engineered binaries for any platform, even x86, we hope that you'll soon feel right at home.

8.1 The Landscape

Architecture
> Von Neumann
> 16-bit words

Registers
> R0: Program Counter
> R1: Stack Pointer
> R2: Status Register
> R3: Constant Generator
> R4-R15: General Use

Address Space
> 16-bit (MSP430)
> 20-bit (MSP430X, X2)

8.2 Memory Map

Unlike other embedded platforms, which like to put the interrupt vector table (IVT) at the beginning of memory, the MSP430 places it at the very end of the 16-bit address space, in Flash. (On smaller chips, this is the very end of Flash.)

Early on, Low RAM at `0x0200` would be the only RAM location, but as that region proved too small, a High RAM area was created at `0x1100`. For firmware compatibility reasons, the Low RAM area is mapped on top of the High RAM area.

Note that Flash grows down from the top of memory, while the RAM grows up. On chips with a 20-bit address space, an Extended Flash region sometimes grows upward from `0x10000`.

Additionally, there is an Info Flash area at `0x1000`. While there is nothing to stop an engineer from using this for code, the region is generally used for configuration settings. In many devices, chips arrive with this region pre-programmed to contain calibration settings for the internal clock.

In most devices, the BSL ROM at `0x0C00` contains a serial bootloader that allows the chip to be reprogrammed even after the JTAG fuse has been blown, and if you know the contents of the last 32 bytes of Flash—the Interrupt Vector Table—you can also read out the contents of memory.

8.3 Loading into a Disassembler

Back in the old days, reverse engineering MSP430 code meant using GNU `objdump` and annotating on pen and paper. Some folks would wrap these tools in Perl, or fill paper notebooks with cross-referencing, but thankfully that's no longer necessary.

Nowadays, IDA Pro has excellent support for the platform. If you have a legit license, just open the Intel Hex image of your target and specify MSP430 as the architecture. Memory locations can be had from the appropriate datasheets.

Radare2's MSP430 support is a bit less mature, and you should make sure to sanity check the disassembly wherever it looks suspect. Luckily, the Radare2 developers are frighteningly quick about fixing bugs, so both bugs that bothered us in the writing this article will likely be patched by the time you read this. For best results, always run Radare2 built from the latest Git repository,[10]—and rebuild it often.

One last tool, which is fast becoming obsolete with Radare2's support, is the MSPGCC project's single-line assembler.[11] It is particularly handy, though, when sanity-checking your own implementation of an assembler or disassembler.

There are no known decompilers for the MSP430, but with small code sizes and rather legible assembly we don't expect one to be necessary.

---

[10]`git clone https://github.com/radare/radare2`
[11]`http://mspgcc.sourceforge.net/assemble.html`

Start	End	Size	Use
0x0000	0x000F	16	Interrupt Control Registers
0x0010	0x00FF	240	8-bit Peripherals
0x0100	0x01FF	255	16-bit Peripherals
0x0200	0x09FF		Low RAM (Mirrored at 0x1100)
0x0C00	0x0FFF	1024	BootStrap Loader (BSL ROM)
0x1000	0x10FF	256	Info Flash
0x1100			High RAM
	0xFFFF		Flash
0x10000			Extended Flash

Table 1 – MSP430 and MSP430X Address Space

8.4 Basics of the Instruction Set

The language is relatively simple, but there are a few dialects that the locals speak. There are 27 action words (instructions), and then some additional emulated instructions which are assembled to one of the 27. Most of these 27 instructions have two forms—.B when they are working on an 8-bit byte, or .W if they want to tackle a 16-bit word. If someone tells you something and doesn't specify it, you can assume it's a word. If you're doing a byte operation in a register, be warned that the most-significant byte is cleared.

The three main types of core words are single-operand arithmetic, two-operand arithmetic, and jumps.

Our simple single-operands are RRC (1-bit rotate right and carry), SWPB (swap the bytes of the word), RRA (1-bit rotate right as arithmetic), SXT (sign-extend a byte into a word), PUSH (onto the stack), CALL (a subroutine, by pushing PC and then moving the new address to PC), and RETI (return from interrupt, restoring the Status Register SR and PC from stack).

Although these are all simple folk, they can, of course, be addressed in many different ways. If our register is n, then we see a few major types of addressing, all based off of the 'As' (for source) and 'Ad' (limited options for destination) fields:

Rn Operate on the contents of register n.

@Rn Operate on what is in memory at the address held in Rn.

@Rn+ Same as above, then increment the register by 1 or 2.[12]

x(Rn) Operate on what is in memory at the address Rn + x.

Wait, we just told you about an 'x'. Where did that come from?! In this case, it's an *extension word*, where the next 16-bit word after the extension defines x. In other words, it's an index off the base address held in Rn.

If the register is r0 (PC, the program counter), r2 (SR, the status register), or r3 (the *constant generator*), special cases apply. A common special case is to give you a constant, either -1, 0, 1, 2, 4, or 8.

Now we tackle two-operand arithmetic operations, most of which you should recognize from any other instruction set. The mov, add, addc (add with carry), sub, and subc instructions are all as you'd expect. cmp pretends to subtract the source from the destination to set status flags. dadd does a decimal addition with carry. xor and and are bitwise operations as usual. We have three that are a little unique: bis (logical OR), bic (dest = dest AND src), and bit (test bits of src AND dest).

Even with these instructions, though, we're still missing many favorite mnemonics that you'll see in disassembly. These are *emulated* instructions, actually implemented using other instruction(s).

For example, br dst (branch) is an emulated instruction. There is no branch opcode, but instead the br instructions are assembled as mov dst, pc. Similarly, pop dst is really mov @SP+, dst, and ret is really mov @sp+, pc. If these mappings make sense, you're all set to continue your travels!

Thus, when we need to get around this land of MSP430, we look not to the many jump types of x86, but instead to simpler patterns, where the only kind of jump operands are relative, and that's that.

---

[12]Here are the rules: Increment by two if registers r0 or r1, or if r4-r15 are used with a .W (2-byte) operand. Increment by 1 if r4 to r15 are used with a .B operand.

So `jmp`, the instruction says, but where to? The first three bits (`001`) mean jump, the next three specify the conditional, and the remaining ten are a signed offset. To get there, the ten bits are multiplied by two (left shifted) and then are added to the program counter, `r0`. Why multiply by two? Well, we have 16-bit word alignment, in the MSP430 land, unlike with those pesky x86 instructions you might be thinking of. Ordnung muß sein!

You might have noticed in your disassembly that even though we told you this was a fixed-width instruction set, some instructions are longer than one 16-bit word! One way this can happen is when using immediate values, which—much like those of the glorious PDP-11 of old—are implemented by dereferencing and incrementing the program counter. This way, the CPU will skip over the immediate value in its code fetch path just as it's fetching that same value as data.

And, finally, there are prefix instructions that have been added in MSP430X, the 20-bit extension of the MSP430. These prefix instructions go before the normal instruction, and you'll most commonly see them setting the upper four bits of the pointer in a 20-bit function call.

8.5 What's a Function, Anyways?

In x86 assembly, we're used to looking for function preambles to pick out the functions—but what do we look for in MSP430 code? We've already discussed finding the entry point of the program and those of other ISRs by looking at the vectors in the IVT. What about other functions?

In MSP430, all functions that are not ISRs will end with a RET instruction—which, as you recall, is actually a `MOV @SP+, PC`.

Compilers vary greatly in the calling conventions—as there is actually no fixed ABI. Usually, arguments get passed in `r12`, `r13`, `r14`, and `r15`. This, however, is by no means a requirement. MSP430 GCC uses `r15` for the first parameter and for most return value types, and `r14`, `r13`, and `r12` for the other parameters. Texas Instruments' Code Composer and the IAR compiler (after EW430 4.10A release) use `r12`, `r13`, `r14`, and `r15` and return in `r12`.

We recommend using an additional heuristic instead of looking for a function preamble format. In this heuristic, we assume that indirect calls are rare, and look for `br #addr` and `call #addr` instructions. Both of these consist of two 16-bit words, and whatever the `#addr` we extract from that second word, there's a good chance that it's the start of a function.

Using this logic, you should be able to find functions even in stripped images disassembled with `msp430-objdump`. A short script, or a good disassembler, should help automate the marking of these functions.

8.6 Making Sense of Interrupts

As with your (other) favorite microcontroller, our exploration of the code can be preempted by an interrupt.

If you don't like these getting in the way of your travels, they can be globally or individually disabled—well, except for the non-maskable interrupts (NMI).[13]

The MSP430 handles any interrupts set in priority order, and goes through the interrupt vector table to find the right interrupt service routine's (ISR) starting address. It hides away the current PC and SR on the stack, and runs the ISR. The ISR then returns, and normal execution continues.

If one thing is for certain, it's that `0xFFFE` is the system's reset ISR address (used on power-up, external reset, etc.), and that it has the highest priority.

If you have an elf32-msp430 formatted dump,[14] use `msp430-objdump dump.msp430 -DS` to get disassembly. Then locate the interrupt table at the end of memory:

```
0000 ffc0  <.sec2 >:
 ffc0 :   26  32   jn  $−946 ; abs  0xfc0e
 ...
 fffc :   26  32   jn  $−946 ; abs  0xfc4a
 fffe :   00  31   jn  $+514 ; abs  0x200
```

We look at `0xFFFE` for the reset interrupt address, which is `0x3100` in this image. That's our entry point into the program, and you can see how it nicely lines up in the disassembly:

```
00003100  <.sec1 >:
 3100:  31  40  00  31    mov     #12544, r1
 3104:  15  42  20  01    mov     &0x0120 , r5
 3108:  75  f3            and.b  #−1,      r5
```

---

[13]Global disable is done by clearing the 'GIE' bit of the status register, `r2`.

[14]If not, use a command like `msp430-objcopy -I ihex -O elf32-msp430 dump.hex dump.msp430` to convert into one.

Maybe we want to look at some specific functionality that is triggered by an interrupt, for example incoming serial data. Looking in the MSP430F1611 data sheet, we find that `USART1` receive is a maskable interrupt at `0xFFE6`. If we look at the notated IVT in an example program (e.g., TinyOS's Printf program compiled for TelosB), we see addresses (in little endian) as shown here:

```
0000ffe0 <__ivtbl_16>:
    ffe0:       52 44       dac/dma
    ffe2:       52 44       i/o p2
    ffe4:       56 56       usart 1 tx
    ffe6:       d0 55       usart 1 rx
    ffe8:       52 44       i/o p1
    ffea:       94 4f       timer a3
    ffec:       76 4f       timer a3
    ffee:       52 44       adc12
    fff0:       52 44       usart 0 tx
    fff2:       52 44       usart 0 rx
    fff4:       52 44       watchdog timer
    fff6:       52 44       compartor a
    fff8:       d8 4f       timer b7
    fffa:       ba 4f       timer b7
    fffc:       52 44       nmi/etc
    fffe:       00 40       reset
```

We note that `0x4452` is used often. A quick look at this address shows that it is an empty IVT noting unused interrupts. Since we're interested in the `USART1` receive path, we follow `0x55d0` and see a large function that in turn calls another function—both nicely annotated, as we were working from an image with debug symbols:

```
000055d0 <sig_UART1RX_VECTOR>:
...
    563a: b0 12 98 46   call #0x4698
...

00004698 <SerialP__rx_state_machine>:
...
```

This technique of looking up your IVT entries and then working backwards to reverse engineer any handlers that correspond to the functionality you are interested in can help you avoid getting lost in reversing unimportant pieces of the code.

8.7 Sorting out Peripherals

If we're reversing some firmware, hopefully we have a target—often this can be data lines going to a radio or some peripheral that carry sensitive data.

Some peripherals are dealt with via interrupts, as shown above, but some are also either partially or totally handled via touching memory defined by the peripheral file map.

In particular, as an alternative to using interrupts, a program could simply poll for incoming data or a change in a pin's state. Likewise, setting up configurations for items such as the `USART` discussed above is done in the peripheral file map.

---

[15]Page 23 of http://www.ti.com/lit/ds/symlink/msp430f1611.pdf

Let us take the same file we used above, and look in the MSP430F1611 guide for the `USART1` in the peripheral file map.[15] Here we see the registers in the range from `0x0078` to `0x007F`. Let us search for a few of these in the image to demonstrate the applicability of this technique.

First, we look for `0x0078` (USART control), `0x0079` (transmit control), and `0x007A` (receive control). We find them all together in a function that is responsible for configuring the USART resource. A reader referencing the documentation will see the other control registers also updated:

```
4e8e <Msp430Uart...Configure...>:
...
4eb4:   c2 4e 78 00   mov.b r14,      &0x0078
4eb8:   d2 42 04 11   mov.b &0x1104,&0x0079
4ebc:   79 00
4ebe:   d2 42 05 11   mov.b &0x1105,&0x007a
4ec2:   7a 00
4ec4:   1e 42 00 11   mov   &0x1100,r14
4ec8:   c2 4e 7c 00   mov.b r14,      &0x007c
4ecc:   8e 10         swpb  r14
4ece:   4e 4e         mov.b r14,      r14
4ed0:   c2 4e 7d 00   mov.b r14,      &0x007d
4ed4:   d2 42 02 11   mov.b &0x1102,&0x007b
...
```

Whereas this approach can help you understand the settings to better sniff the serial bus physically, often you'd rather want to understand the actual data being written out. For this, we look for the peripheral holding the transmit buffer pointer—in our case at `0x007F`, according to the chip documentation. Searching for this in the disassembly leads us to a few interesting functions. Firstly, there's one that disables the UART, which fills this address with null bytes. This helps us confirm we're looking at the right address. We also see this address written to in the interrupt handler that we located in the previous section—and in a large function that ends up being a form of `printf` for writing out to this serial line.

As you can see, working backwards from the addresses located in the peripheral file map can help you quickly find functions of interest.

— — — —   — — —   — — — —

This guide is neither complete nor perfectly accurate. We told a few lies-to-children as all teachers do, and we omitted a dozen nifty examples that would've fit. Still, we hope that this will whet your appetite for working with the MSP430 architecture, and that, when you begin to work on the '430s, you can get your bearings quickly, jumping into the fun part of the journey with less hassle.

Also, for more MSP430 exploitation tricks, check out PoC‖GTFO 2:5!
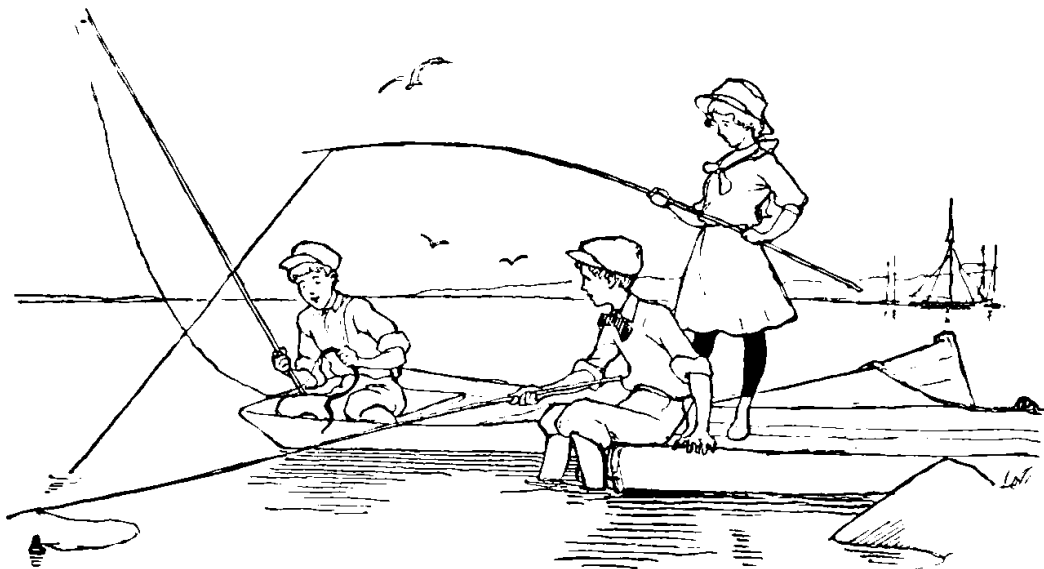
9 This HTML page is also a PDF which is also a ZIP which is also a Ruby script which is an HTTP quine; or, The Treachery of Files

by Evan Sultanik
from a concept independently conceived by Ange Albertini
and with great technical assistance from Philippe Teuwen

Please rise and open your hymnal for the recitation of PoC‖GTFO 7:6.

> "A file has no intrinsic meaning. The meaning of a file—its type, its validity, its contents—can be different for each parser or interpreter. "

You may be seated.

In the spirit of самиздат and the license of this publication, we thought it might be nifty to aid its promulgation by enabling the PDF to mirror itself. That's right, this PDF is an HTTP quine: it is a web server that serves copies of itself.

```
$ ruby pocorgtfo11.pdf &
Listening for connections on port 8080.
To listen on a different port,
re-run with the desired port as a command-line argument.
$ curl -s http://localhost:8080/pocorgtfo11.pdf | diff -s - pocorgtfo11.pdf
A neighbor at 127.0.0.1 is requesting /pocorgtfo11.pdf
Files - and pocorgtfo11.pdf are identical
```
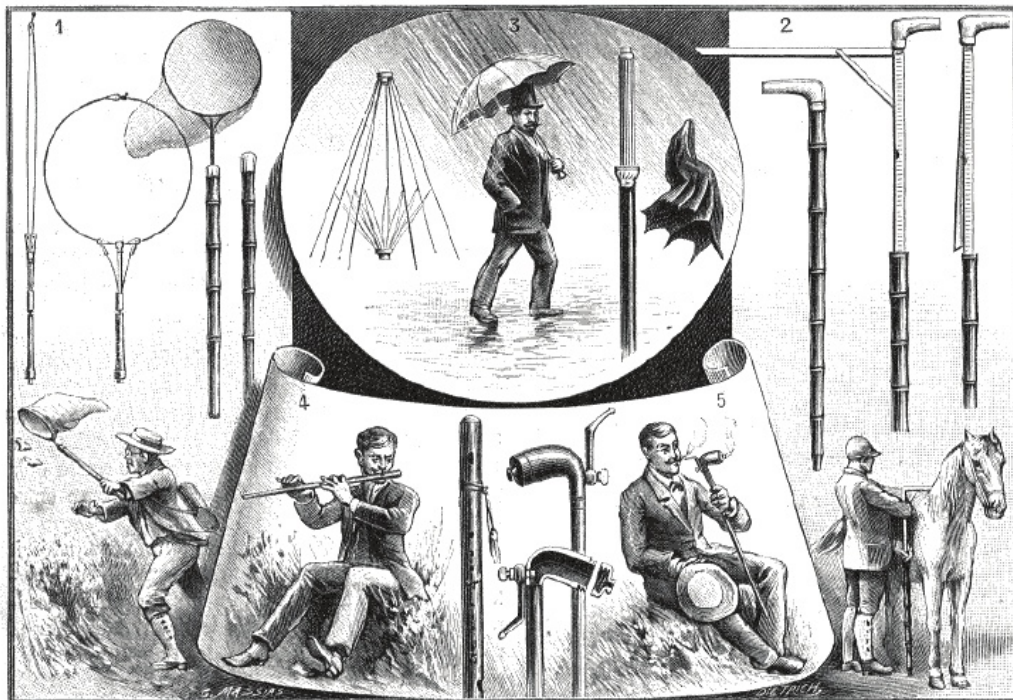

Utilisation de la canne. — 1. Canne-filet à papillons. — 2. Canne à toiser les chevaux. — 3. Canne-parapluie. — 4. Canne musicale. — 5. Ceci n'est pas une pipe.

This polyglot once again exploits the fact that PDF readers ignore everything before the first instance of "%PDF". Coupled with Ruby's `__END__` token—which effectively halts interpretation—and its `__FILE__` token—which resolves to the path of the file being interpreted—it's actually quite easy to make an HTTP quine by prepending the PDF with the following:

```ruby
require 'socket'
server = TCPServer.new('', 8080)
loop do
        socket = server.accept
        request = socket.gets
        response = File.open(__FILE__).read
        socket.print "HTTP/1.1 200 OK\r\n" +
                "Content-Type: application/
pdf\r\n" +
                "Content-Length: #{response.
bytesize}\r\n" +
                "Connection: close\r\n"
        socket.print "\r\n"
        socket.print response
        socket.close
end
__END__
```

But why stop there? Ruby makes all of the bytes in the script that occur after the `__END__` token available in the special "`DATA`" object. Therefore, we can add additional content between `__END__` and `%PDF` that the script can serve.

```ruby
require 'socket'
server = TCPServer.new('', 8080)
html = DATA.read().split(/<\/html>/)[0]+"</
    html>\n"
loop do
        socket = server.accept
        if socket.gets.split(' ')[1].
    downcase.end_with? ".pdf" then
                c = "application/pdf"
                d = File.open(__FILE__).read
                n = File.size(__FILE__)
        else
                c = "text/html"
                d = html
                n = html.length
        end
        socket.print "HTTP/1.1 200 OK\r\
    nContent-Type: #{c}\r\nContent-Length:
    #{n}\r\nConnection: close\r\n\r\n"+d
        socket.close
end
__END__
<html>
    <head>
    <title>An HTTP Quine PoC</title>
    </head>
    <body>
      <a href="pocorgtfo11.pdf">Download
      pocorgtfo11.pdf!</a>
```

```
        </body>
</html>
```

Any HTTP request with a URL that ends with `.pdf` will result in a copy of the PDF; anything else will result in the HTML index parsed from `DATA`.

Since the data between `__END__` and `%PDF...` is pure HTML already, it would be a shame not to make this file a pure HTML polyglot, too (similar to PoC‖GTFO 0x07). Doing so is relatively simple by wrapping PDF in HTML comments:

```
INSERT RUBY WEB SERVER HERE
__END__
<html>
    ...
</html>
<!--
INSERT RAW PDF HERE
-->
```

This is valid Ruby, since Ruby does not interpret anything after the `__END__`. The PDF does not affect the validity of the HTML since it is commented. There will be trouble if the byte sequence "`-->`" (2D 2D 3E) occurs anywhere within the PDF, but this is very unlikely and has proven not to be a problem.

Wrapping the Ruby webserver code in an HTML comment would have been ideal, and does in fact work for most PDF viewers. However, the presence of an HTML opening comment before the `%PDF` causes Adobe's parser to classify the file as HTML and therefore refuse to open it.

Unfortunately, some web browsers interpret the Ruby code as having an implied "`<html>`" preceding it, adding all of that text to the DOM. This is remedied with Javascript in the HTML that sanitizes the DOM if necessary.

As has become the norm, this PDF is also a valid ZIP. This feat does not affect the Ruby/HTML portion since the ZIP is embedded later in the file as an object within the PDF (cf. PoC‖GTFO 1:5). This presents an additional opportunity for the webserver: if the script can unzip itself, then it can also serve all of the contents of the ZIP. Unfortunately, Ruby does not have a ZIP decompression facility in its standard library. Therefore, the webserver calls the `unzip` utility with the "`-l`" option, parsing the output to determine the names and sizes of the constituent files. Then, a call to `unzip` with "`-p`" writes raw decompressed contents to STDOUT, which the web server splits apart and stores in memory. Any HTTP request with a URL that matches a

34

file path within the ZIP is served that decompressed file. This allows us to have images like a `favicon` in the HTML. In the event that the PDF is interpreted as raw HTML—*i.e.*, it was *not* served from the Ruby script—a Javascript function conveniently hides all of the ZIP access portions.

With all of this feature bloat, the Ruby/HTML code that is prepended before the PDF started getting quite large. Unfortunately, some PDF readers like PDFium[16] (the default PDF viewer shipped with Chrom(e|ium)) fail unless they find "`%PDF`" within the first 1024 characters. Therefore, the final trick in this polyglot is to exploit Ruby's multiline comment syntax (which, like the `__END__` token, owes itself to Ruby's Perl heritage). This allows us to start the PDF header early, within a comment that will not be interpreted. Within that PDF header we open a dummy object stream that will contain the remainder of the Ruby script and the following HTML code before the start of the "real" PDF.

```
   require 'socket'
2  =begin
   %PDF-1.5
4  9999 0 obj
   <<
6  /Length INSERT_#
       _REMAINING_RUBY_AND_HTML_BYTES_HERE
   >>
8  stream
   =end
10 INSERT REMAINING RUBY CODE HERE
       __END__
12 INSERT HTML HERE
   <!--
14 endstream
   endobj
16 INSERT RAW PDF HERE WITH LEADING %... HEADER
       REMOVED
   -->
```

Figure 5 describes the anatomy of the polyglot, as interpreted in each file format.

---

[16]https://pdfium.googlesource.com/pdfium/

Ruby · HTML · PDF · ZIP

Ruby

```
require statements
```
```
=begin
```
> Multiline Comment

```
=end
```

Ruby Webserver
Parses the HTML from `DATA` and calls `unzip` on itself to extract the ZIP content

```
__END__
```

> Everything after `__END__` is accessible from Ruby's special `DATA` object

HTML

> Text occurring before `<html>`. Some browsers will add this to the DOM, ignoring the following `<html>` and `<head>`.

HTML
Javascript to remove everything between "require…" and "__END__" from the DOM, if necessary

```
<!--
```

```
-->
```

PDF

```
PDF Header
```
```
9999 0 obj
<<
/Length  ?
>>

stream
```

> Replace ? with the number of bytes here (*i.e.*, between `stream` and `endstream`)

```
endstream
endobj
```
PDF Content
```
obj/stream
```

```
endstream/endobj
```
```
PDF Footer
```

ZIP

ZIP Content
as usual
(cf. PoC‖GTFO 1:5 and 9:12)

```
Central Directory
```
```
Archive Comment
```

Figure 5 – Anatomy of the Ruby/HTML/PDF/ZIP polyglot. Green portions contain the main content of their respective filetypes. White portions are for context and to illustrate modifications necessary to make the polyglot work. Gray portions are not interpreted by their respective filetypes.

E. E. *or* PHYSICS GRADUATES

with experience in

RADAR *or* ELECTRONICS

*or those desiring
to enter these areas...*
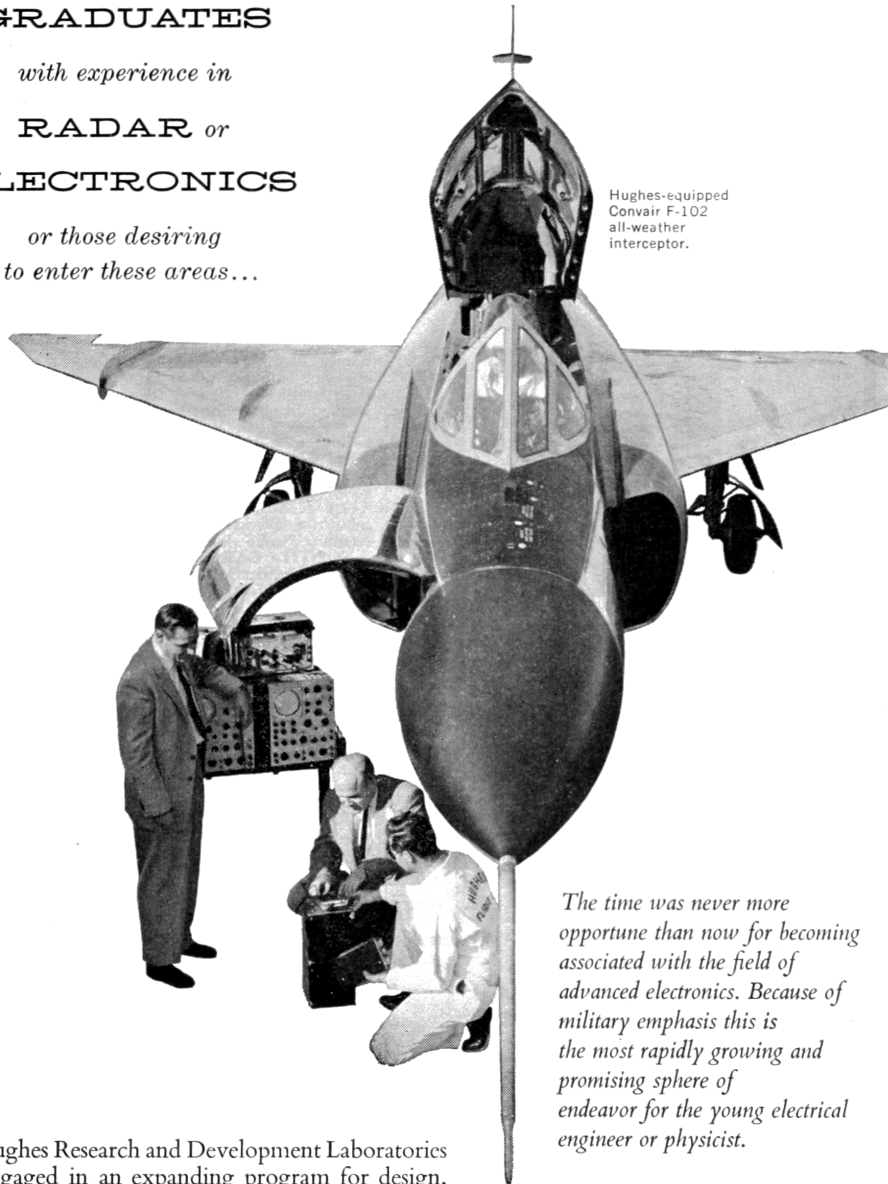
Hughes-equipped
Convair F-102
all-weather
interceptor.

*The time was never more
opportune than now for becoming
associated with the field of
advanced electronics. Because of
military emphasis this is
the most rapidly growing and
promising sphere of
endeavor for the young electrical
engineer or physicist.*

Since 1948 Hughes Research and Development Laboratories have been engaged in an expanding program for design, development and manufacture of highly complex radar fire control systems for fighter and interceptor aircraft. This requires Hughes technical advisors in the field to serve companies and military agencies employing the equipment.

As one of these field engineers *you will become familiar with the entire systems* involved, including the most advanced electronic computers. With this advantage you will be ideally situated to broaden your experience and learning more quickly for future application to advanced electronics activity in either the military or the commercial field.

Positions are available in the continental United States for married and single men under 35 years of age. Overseas assignments are open to single men only.

SCIENTIFIC AND
ENGINEERING STAFF

HUGHES

RESEARCH AND
DEVELOPMENT
LABORATORIES

*Culver City,
Los Angeles County,
California*

Relocation of applicant must not cause
disruption of an urgent military project.

37

10 In Memoriam: Ben "bushing" Byer

by fail0verflow

Ben Byer
1980–2016

We are deeply saddened by the news that our member, colleague, and friend Ben "bushing" Byer passed away of natural causes on Monday, February 8th.

Many of you knew him as one of the public faces of our group, fail0verflow, and before that, Team Twiizers and the iPhone Dev Team.

Outspoken but never confrontational, he was proof that even in the competitive and often aggressive hacking scene, there is a place for both a sharp mind and a kind heart.

To us he was, of course, much more. He brought us together, as a group and in spirit. Without him, we as a team would not exist. He was a mentor to many, and an inspiration to us all.

Yet above anything, he was our friend. He will be dearly missed.

Our thoughts go out to his wife and family.

Keep hacking. It's what bushing would have wanted.
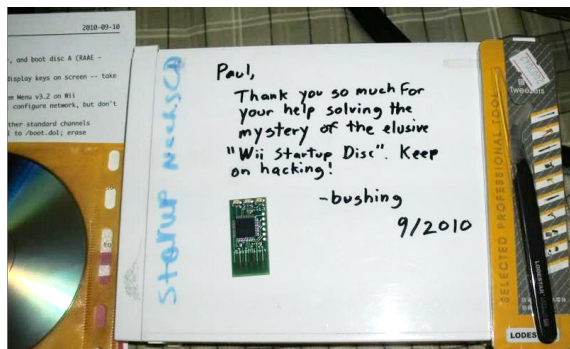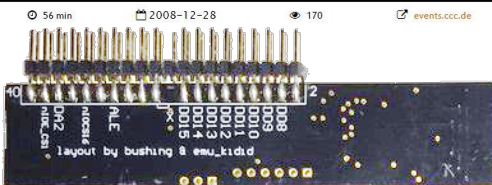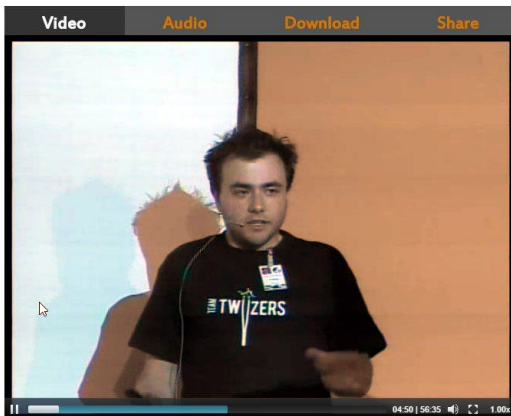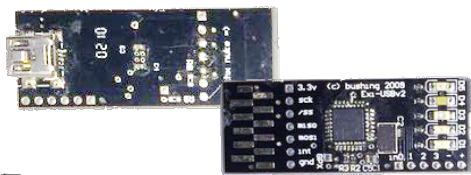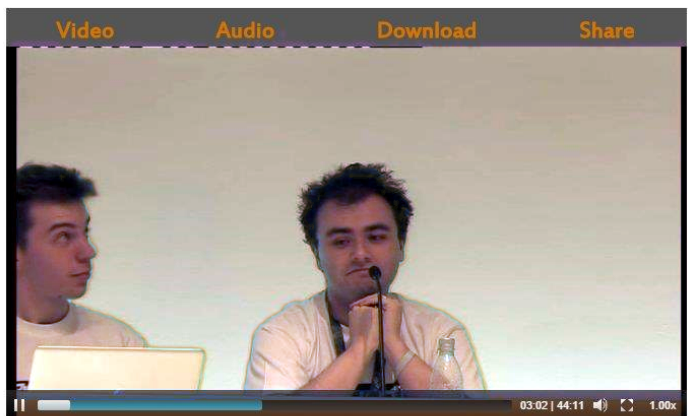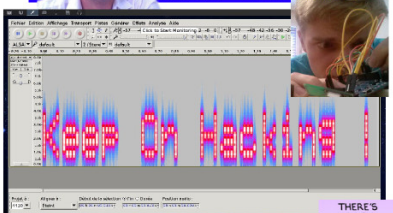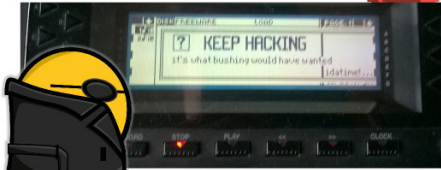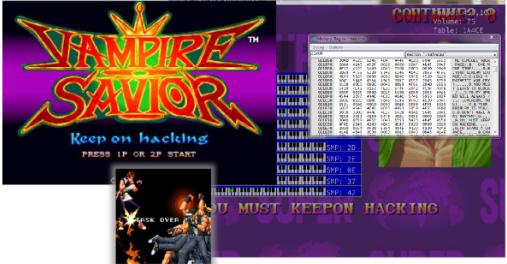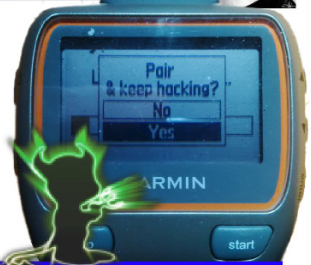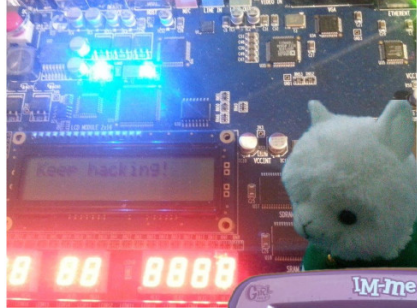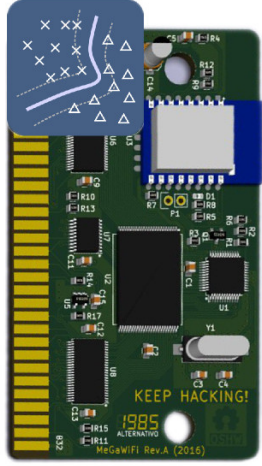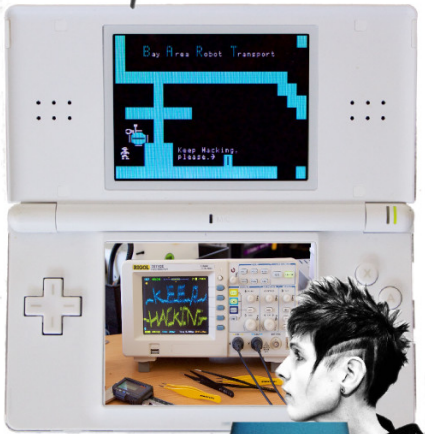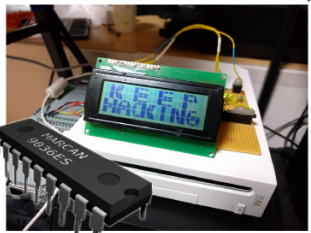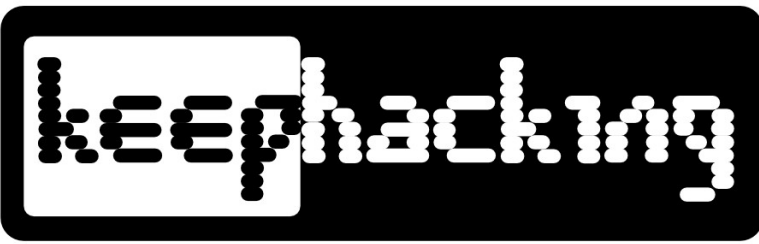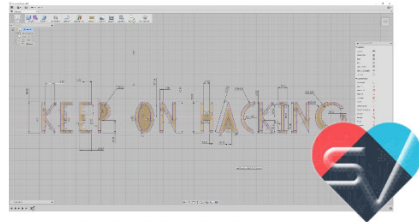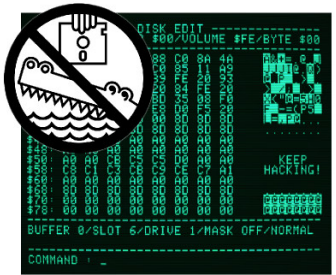
11 Tithe us your Alms of 0day!

by Pastor Manul Laphroaig,
Unlicensed Proselytizer
International Church of the Weird Machines

Howdy, neighbor!

A man came to me, and he said, "Forgive me, Preacher, for I have sinned. I play piano in a brothel."

I laughed, "That ain't no sin, neighbor. Folks need their music. Go now in peace."

But the man was worried, he said, "No, Preacher, I've really sinned. I need your forgiveness."

So I laughed again, "Go now, you are forgiven! Stop wasting my time."

"But Preacher, I teach children to use PHP!"

"Why would you lie to me about your profession like that?"

"Oh, *you* try confessing an occupation like that!"

"I'm glad I don't have to," I said while finishing my drink, "'cause until today I didn't believe there was any fate I feared more than hell."

Do this: write an email telling our editors how to do reproduce *ONE* clever, technical trick from your research. If you are uncertain of your English, we'll happily translate from French, Russian, Southern Appalachian, and German. If you don't speak those languages, we'll draft a translator from those poor sods who owe us favors.

Like an email, keep it short. Like an email, you should assume that we already know more than a bit about hacking, and that we'll be insulted or—WORSE!—that we'll be bored if you include a long tutorial where a quick reminder would do.

Just use 7-bit ASCII if your language doesn't require funny letters, as whenever we receive something typeset in OpenOffice, we briefly mistake it for a ransom note. Don't try to make it thorough or broad. Don't use bullet-points, as this isn't a damned Powerpoint deck. Keep your code samples short and sweet; we can leave the long-form code as an attachment. Do not send us LaTeX; it's our job to do the typesetting!

Do pick one quick, clever trick and explain it in a few pages. Teach me how to write a memory-corruption exploit—not just shellcode!–that triggers the same bug without profiling on MIPS, PowerPC, x86, and AMD64. Show me how to write a 64-bit DOS extender, or how to extract firmware from locked regions on an MSP432's funky flash protection.

Don't tell me that it's possible; rather, teach me how to do it myself with the absolute minimum of formality and bullshit.

Like an email, we expect informal (or faux-biblical) language and hand-sketched diagrams. Write it in a single sitting, and leave any editing for your poor preacherman to do over a bottle of fine scotch. Send this to pastor@phrack.org and hope that the neighborly Phrack folks—praise be to them!—aren't man-in-the-middling our submission process.

Yours in PoC and Pwnage,
Pastor Manul Laphroaig, D.D.